axiom™

The 30 Year Horizon

Manuel Bronstein   William Burge   Timothy Daly
James Davenport   Michael Dewar   Martin Dunstan
Albrecht Fortenbacher Patrizia Gianni   Johannes Grabmeier
Jocelyn Guidry   Richard Jenks   Larry Lambe
Michael Monagan   Scott Morrison   William Sit
Jonathan Steinbach   Robert Sutor   Barry Trager
Stephen Watt   Jim Wen   Clifton Williamson

Contents

New Foreword

On October 1, 2001 Axiom was withdrawn from the market and ended life as a commercial product. On September 3, 2002 Axiom was released under the Modified BSD license, including this document. On August 27, 2003 Axiom was released as free and open source software available for download from the Free Software Foundation's website, Savannah.

Work on Axiom has had the generous support of the Center for Algorithms and Interactive Scientific Computation (CAISS) at City College of New York. Special thanks go to Dr. Gilbert Baumslag for his support of the long term goal.

This document is a complete "re-implementation" of the original Axiom book by Jenks and Sutor. Virtually every line has been reviewed and rewritten into the new Axiom pamphlet format. Changes were made to reflect the new Axiom system. Additional material was added and some previous examples were rewritten. This is intended to be a "living" document with material referenced or gathered automatically from other parts of the system documentation. Future plans include adding active examples (moving graphics, in-line command prompts) using Active-DVI.

Axiom has been in existence for over thirty years. It is estimated to contain about three hundred man-years of research and has, as of September 3, 2003, 143 people listed in the credits. All of these people have contributed directly or indirectly to making Axiom available. Axiom is being passed to the next generation. I'm looking forward to future milestones.

With that in mind I've introduced the theme of the "30 year horizon". We must invent the tools that support the Computational Mathematician working 30 years from now. How will research be done when every bit of mathematical knowledge is online and instantly available? What happens when we scale Axiom by a factor of 100, giving us 1.1 million domains? How can we integrate theory with code? How will we integrate theorems and proofs of the mathematics with space-time complexity proofs and running code? What visualization tools are needed? How do we support the conceptual structures and semantics of mathematics in effective ways? How do we support results from the sciences? How do we teach the next generation to be effective Computational Mathematicians?

The "30 year horizon" is much nearer than it appears.

Tim Daly
CAISS, City College of New York
November 10, 2003 ((iHy))

Foreword

You are holding in your hands an unusual book. Winston Churchill once said that the empires of the future will be empires of the mind. This book might hold an electronic key to such an empire.

When computers were young and slow, the emerging computer science developed dreams of Artificial Intelligence and Automatic Theorem Proving in which theorems can be proved by machines instead of mathematicians. Now, when computer hardware has matured and become cheaper and faster, there is not too much talk of putting the burden of formulating and proving theorems on the computer's shoulders. Moreover, even in those cases when computer programs do prove theorems, or establish counter-examples (for example, the solution of the four color problem, the non-existence of projective planes of order 10, the disproof of the Mertens conjecture), humans carry most of the burden in the form of programming and verification.

It is the language of computer programming that has turned out to be the crucial instrument of productivity in the evolution of scientific computing. The original Artificial Intelligence efforts gave birth to the first symbolic manipulation systems based on LISP. The first complete symbolic manipulation or, as they are called now, computer algebra packages tried to imbed the development programming and execution of mathematical problems into a framework of familiar symbolic notations, operations and conventions. In the third decade of symbolic computations, a couple of these early systems—REDUCE and MACSYMA—still hold their own among faithful users.

Axiom was born in the mid-70's as a system called Scratchpad developed by IBM researchers. Scratchpad/Axiom was born big—its original platform was an IBM mainframe 3081, and later a 3090. The system was growing and learning during the decade of the 80's, and its development and progress influenced the field of computer algebra. During this period, the first commercially available computer algebra packages for mini and and microcomputers made their debut. By now, our readers are aware of Mathematica, Maple, Derive, and Macsyma. These systems (as well as a few special purpose computer algebra packages in academia) emphasize ease of operation and standard scientific conventions, and come with a prepared set of mathematical solutions for typical tasks confronting an applied scientist or an engineer. These features brought a recognition of the enormous benefits of computer algebra to the widest circles of scientists and engineers.

The Scratchpad system took its time to blossom into the beautiful Axiom product. There is no rival to this powerful environment in its scope and, most importantly, in its structure and organization. Axiom contains the basis for any comprehensive and elaborate mathematical development. It gives the user all Foundation and Algebra instruments necessary to develop a computer realization of sophisticated mathematical objects in exactly the way a mathematician

would do it. Axiom is also the basis of a complete scientific cyberspace—it provides an environment for mathematical objects used in scientific computation, and the means of controlling and communicating between these objects. Knowledge of only a few Axiom language features and operating principles is all that is required to make impressive progress in a given domain of interest. The system is powerful. It is not an interactive interpretive environment operating only in response to one line commands—it is a complete language with rich syntax and a full compiler. Mathematics can be developed and explored with ease by the user of Axiom. In fact, during Axiom's growth cycle, many detailed mathematical domains were constructed. Some of them are a part of Axiom's core and are described in this book. For a bird's eye view of the algebra hierarchy of Axiom, glance inside the book cover.

The crucial strength of Axiom lies in its excellent structural features and unlimited expandability—it is open, modular system designed to support an ever growing number of facilities with minimal increase in structural complexity. Its design also supports the integration of other computation tools such as numerical software libraries written in FORTRAN and C. While Axiom is already a very powerful system, the prospect of scientists using the system to develop their own fields of Science is truly exciting—the day is still young for Axiom.

Over the last several years Scratchpad/Axiom has scored many successes in theoretical mathematics, mathematical physics, combinatorics, digital signal processing, cryptography and parallel processing. We have to confess that we enjoyed using Scratchpad/Axiom. It provided us with an excellent environment for our research, and allowed us to solve problems intractable on other systems. We were able to prove new diophantine results for π; establish the Grothendieck conjecture for certain classes of linear differential equations; study the arithmetic properties of the uniformization of hyperelliptic and other algebraic curves; construct new factorization algorithms based on formal groups; within Scratchpad/Axiom we were able to obtain new identities needed for quantum field theory (elliptic genus formula and double scaling limit for quantum gravity), and classify period relations for CM varieties in terms of hypergeometric series.

The Axiom system is now supported and distributed by NAG, the group that is well known for its high quality software products for numerical and statistical computations. The development of Axiom in IBM was conducted at IBM T.J. Watson Research Center at Yorktown, New York by a symbolic computation group headed by Richard D. Jenks. Shmuel Winograd of IBM was instrumental in the progress of symbolic research at IBM.

This book opens the wonderful world of Axiom, guiding the reader and user through Axiom's definitions, rules, applications and interfaces. A variety of fully developed areas of mathematics are presented as packages, and the user is well advised to take advantage of the sophisticated realization of familiar mathematics. The Axiom book is easy to read and the Axiom system is easy to use. It possesses all the features required of a modern computer environment (for example, windowing, integration of operating system features, and interactive

graphics). Axiom comes with a detailed hypertext interface (HyperDoc), an elaborate browser, and complete on-line documentation. The HyperDoc allows novices to solve their problems in a straightforward way, by providing menus for step-by-step interactive entry.

The appearance of Axiom in the scientific market moves symbolic computing into a higher plane, where scientists can formulate their statements in their own language and receive computer assistance in their proofs. Axiom's performance on workstations is truly impressive, and users of Axiom will get more from them than we, the early users, got from mainframes. Axiom provides a powerful scientific environment for easy construction of mathematical tools and algorithms; it is a symbolic manipulation system, and a high performance numerical system, with full graphics capabilities. We expect every (computer) power hungry scientist will want to take full advantage of Axiom.

David V. Chudnovsky   Gregory V. Chudnovsky

Introduction to Axiom

0.1 Introduction to Axiom

Welcome to the world of Axiom. We call Axiom a scientific computation system:
a self-contained toolbox designed to meet your scientific programming needs,
from symbolics, to numerics, to graphics.

This introduction is a quick overview of what Axiom offers.

0.1.1 Symbolic Computation

Axiom provides a wide range of simple commands for symbolic mathematical
problem solving. Do you need to solve an equation, to expand a series, or to
obtain an integral? If so, just ask Axiom to do it.

Given

$$\int \left(\frac{1}{\left(x^3 \ (a+bx)^{1/3}\right)} \right) dx$$

we would enter this into Axiom as:

```
integrate(1/(x**3 * (a+b*x)**(1/3)),x)
```

which would give the result:

$$
\frac{\left(\begin{array}{l}
-2 \ b^2 \ x^2 \ \sqrt{3} \ \log\left(\sqrt[3]{a} \ \sqrt[3]{b \ x + a}^2 + \sqrt[3]{a}^2 \ \sqrt[3]{b \ x + a} + a\right) + \\[2ex]
4 \ b^2 \ x^2 \ \sqrt{3} \ \log\left(\sqrt[3]{a}^2 \ \sqrt[3]{b \ x + a} - a\right) + \\[2ex]
12 \ b^2 \ x^2 \ \arctan\left(\dfrac{2 \ \sqrt{3} \ \sqrt[3]{a}^2 \ \sqrt[3]{b \ x + a} + a \ \sqrt{3}}{3 \ a}\right) + \\[2ex]
(12 \ b \ x - 9 \ a) \ \sqrt{3} \ \sqrt[3]{a} \ \sqrt[3]{b \ x + a}^2
\end{array}\right)}{18 \ a^2 \ x^2 \ \sqrt{3} \ \sqrt[3]{a}}
$$

<div align="center">Type: Union(Expression Integer,...)</div>

Axiom provides state-of-the-art algebraic machinery to handle your most ad-
vanced symbolic problems. For example, Axiom's integrator gives you the an-
swer when an answer exists. If one does not, it provides a proof that there is

no answer. Integration is just one of a multitude of symbolic operations that Axiom provides.

0.1.2 Numeric Computation

Axiom has a numerical library that includes operations for linear algebra, solution of equations, and special functions. For many of these operations, you can select any number of floating point digits to be carried out in the computation.

Solve $x^{49} - 49x^4 + 9$ to 49 digits of accuracy. First we need to change the default output length of numbers:

```
digits(49)
```

and then we execute the command:

```
solve(x**49-49*x**4+9 = 0,1.e-49)
```

$$[x = -0.6546536706904271136718122105095984761851224331556,$$

$$x = 1.0869213956538595084939390359548932890092133388763,$$

$$x = 0.6546536707255271739694686066136764835361487607661]$$

                                    Type: List Equation Polynomial Float

The output of a computation can be converted to FORTRAN to be used in a later numerical computation. Besides floating point numbers, Axiom provides literally dozens of kinds of numbers to compute with. These range from various kinds of integers, to fractions, complex numbers, quaternions, continued fractions, and to numbers represented with an arbitrary base.

What is 10 to the 90-th power in base 32?

```
radix(10**90,32)
```

returns:

```
FMM3O955CSEIV0ILKH82OCN3I7PICQU0OQMD0FV6TP000000000000000000
```

                                    Type: RadixExpansion 32

The AXIOM numerical library can be enhanced with a substantial number of functions from the NAG library of numerical and statistical algorithms. These

functions will provide coverage of a wide range of areas including roots of functions, Fourier transforms, quadrature, differential equations, data approximation, non-linear optimization, linear algebra, basic statistics, step-wise regression, analysis of variance, time series analysis, mathematical programming, and special functions. Contact the Numerical Algorithms Group Limited, Oxford, England.

0.1.3 Graphics

You may often want to visualize a symbolic formula or draw a graph from a set of numerical values. To do this, you can call upon the Axiom graphics capability. Draw $J_0(\sqrt{x^2 + y^2})$ for $-20 \leq x, y \leq 20$.

```
draw(5*besselJ(0,sqrt(x**2+y**2)), x=-20..20, y=-20..20)
```
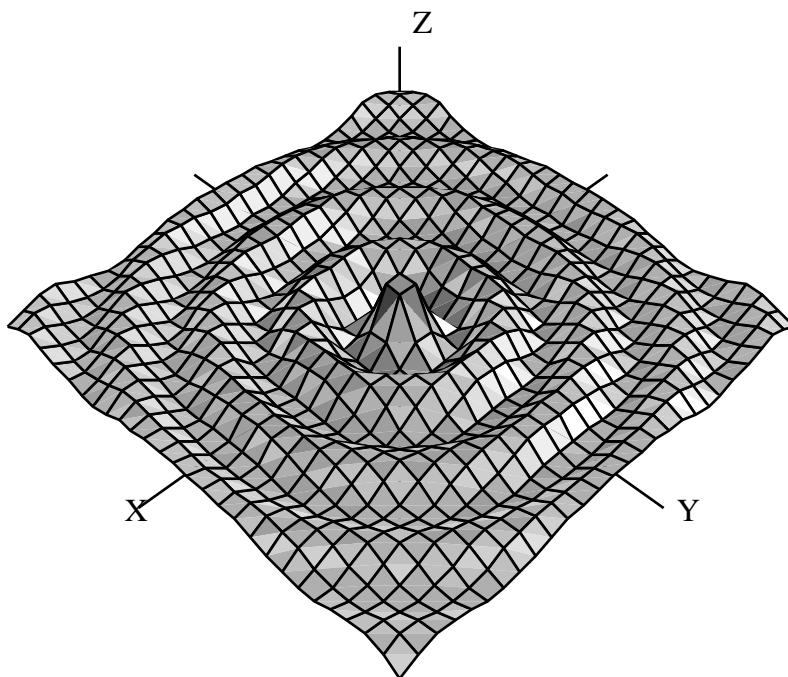

Figure 1: $J_0(\sqrt{x^2 + y^2})$ for $-20 \leq x, y \leq 20$

Graphs in Axiom are interactive objects you can manipulate with your mouse. Just click on the graph, and a control panel pops up. Using this mouse and the control panel, you can translate, rotate, zoom, change the coloring, lighting, shading, and perspective on the picture. You can also generate a PostScript copy of your graph to produce hard-copy output.

0.1.4   HyperDoc
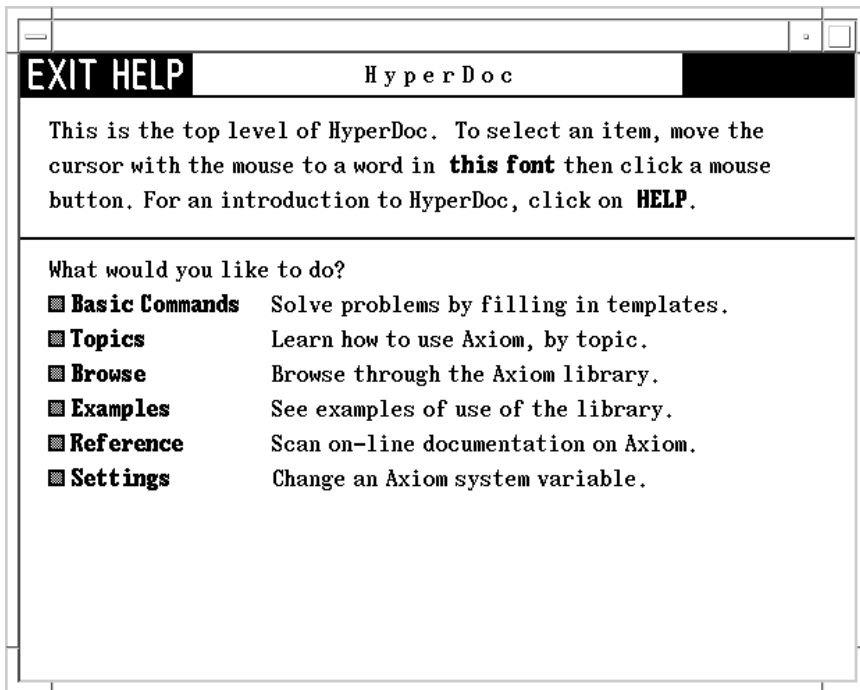
Figure 2: Hyperdoc opening menu

HyperDoc presents you windows on the world of Axiom, offering on-line help, examples, tutorials, a browser, and reference material. HyperDoc gives you on-line access to this document in a "hypertext" format. Words that appear in a different font (for example, `Matrix`, **factor**, and *category*) are generally mouse-active; if you click on one with your mouse, HyperDoc shows you a new window for that word.

As another example of a HyperDoc facility, suppose that you want to compute the roots of $x^{49} - 49x^4 + 9$ to 49 digits (as in our previous example) and you don't know how to tell Axiom to do this. The "basic command" facility of HyperDoc leads the way. Through the series of HyperDoc windows shown in Figure 2 on page 4 and the specified mouse clicks, you and HyperDoc generate the correct command to issue to compute the answer.

0.1.5 Interactive Programming

Axiom's interactive programming language lets you define your own functions. A simple example of a user-defined function is one that computes the successive Legendre polynomials. Axiom lets you define these polynomials in a piece-wise way.

The first Legendre polynomial.

```
p(0) == 1
```

                                                           Type: Void

The second Legendre polynomial.

```
p(1) == x
```

                                                            Type: Void

The n-th Legendre polynomial for $(n > 1)$.

```
p(n) == ((2*n-1)*x*p(n-1) - (n-1) * p(n-2))/n
```

                                                            Type: Void

In addition to letting you define simple functions like this, the interactive language can be used to create entire application packages. All the graphs in the Axiom images section were created by programs written in the interactive language.

The above definitions for p do no computation—they simply tell Axiom how to compute $p(k)$ for some positive integer k.

To actually get a value of a Legendre polynomial, you ask for it.

What is the tenth Legendre polynomial?

```
p(10)
```

```
   Compiling function p with type Integer -> Polynomial Fraction
      Integer
   Compiling function p as a recurrence relation.
```

$$\frac{46189}{256}\,x^{10} - \frac{109395}{256}\,x^8 + \frac{45045}{128}\,x^6 - \frac{15015}{128}\,x^4 + \frac{3465}{256}\,x^2 - \frac{63}{256}$$

<div align="right">Type: Polynomial Fraction Integer</div>

Axiom applies the above pieces for p to obtain the value of $p(10)$. But it does more: it creates an optimized, compiled function for p. The function is formed by putting the pieces together into a single piece of code. By *compiled*, we mean that the function is translated into basic machine-code. By *optimized*, we mean that certain transformations are performed on that code to make it run faster. For p, Axiom actually translates the original definition that is recursive (one that calls itself) to one that is iterative (one that consists of a simple loop).

What is the coefficient of x^{90} in $p(90)$?

```
coefficient(p(90),x,90)
```

$$\frac{5688265554205201782222345823742658185356149744909 5175}{773712524553362671811195264}$$

<div align="right">Type: Polynomial Fraction Integer</div>

In general, a user function is type-analyzed and compiled on first use. Later, if you use it with a different kind of object, the function is recompiled if necessary.

0.1.6 Data Structures

A variety of data structures are available for interactive use. These include strings, lists, vectors, sets, multisets, and hash tables. A particularly useful structure for interactive use is the infinite stream:

Create the infinite stream of derivatives of Legendre polynomials.

```
[D(p(i),x) for i in 1..]
```

$$\Bigg[1, 3\,x, \frac{15}{2}\,x^2 - \frac{3}{2}, \frac{35}{2}\,x^3 - \frac{15}{2}\,x, \frac{315}{8}\,x^4 - \frac{105}{4}\,x^2 + \frac{15}{8},$$

$$\frac{693}{8}\,x^5 - \frac{315}{4}\,x^3 + \frac{105}{8}\,x, \frac{3003}{16}\,x^6 - \frac{3465}{16}\,x^4 + \frac{945}{16}\,x^2 - \frac{35}{16},$$

$$\frac{6435}{16}\,x^7 - \frac{9009}{16}\,x^5 + \frac{3465}{16}\,x^3 - \frac{315}{16}\,x,$$

$$\frac{109395}{128}\,x^8 - \frac{45045}{32}\,x^6 + \frac{45045}{64}\,x^4 - \frac{3465}{32}\,x^2 + \frac{315}{128},$$

$$\frac{230945}{128}\,x^9 - \frac{109395}{32}\,x^7 + \frac{135135}{64}\,x^5 - \frac{15015}{32}\,x^3 + \frac{3465}{128}\,x, \dots\Bigg]$$

Streams display only a few of their initial elements. Otherwise, they are "lazy": they only compute elements when you ask for them.

Data structures are an important component for building application software. Advanced users can represent data for applications in optimal fashion. In all, Axiom offers over forty kinds of aggregate data structures, ranging from mutable structures (such as cyclic lists and flexible arrays) to storage efficient structures (such as bit vectors). As an example, streams are used as the internal data structure for power series.

What is the series expansion of $\log(\cot(x))$ about $x = \pi/2$?

```
series(log(cot(x)),x = %pi/2)
```

$$\log\left(\frac{-2\,x + \pi}{2}\right) + \frac{1}{3}\left(x - \frac{\pi}{2}\right)^2 + \frac{7}{90}\left(x - \frac{\pi}{2}\right)^4 + \frac{62}{2835}\left(x - \frac{\pi}{2}\right)^6 +$$

$$\frac{127}{18900}\left(x - \frac{\pi}{2}\right)^8 + \frac{146}{66825}\left(x - \frac{\pi}{2}\right)^{10} + O\left(\left(x - \frac{\pi}{2}\right)^{11}\right)$$

Type: GeneralUnivariatePowerSeries(Expression Integer,x,pi/2)

Series and streams make no attempt to compute *all* their elements! Rather, they stand ready to deliver elements on demand.

What is the coefficient of the 50-th term of this series?

```
coefficient(%,50)
```

$$\frac{44590788901016030052447242300856550965644}{71314692864386691115840908813093603545813591330859375}$$

Type: Expression Integer

0.1.7 Mathematical Structures

Axiom also has many kinds of mathematical structures. These range from simple ones (like polynomials and matrices) to more esoteric ones (like ideals and Clifford algebras). Most structures allow the construction of arbitrarily complicated "types."

Even a simple input expression can result in a type with several levels.

```
matrix [ [x + %i,0], [1,-2] ]
```

$$\begin{bmatrix} x+i & 0 \\ 1 & -2 \end{bmatrix}$$

<div align="center">Type: Matrix Polynomial Complex Integer</div>

The Axiom interpreter builds types in response to user input. Often, the type of the result is changed in order to be applicable to an operation.

The inverse operation requires that elements of the above matrices are fractions.

```
inverse(%)
```

$$\begin{bmatrix} \frac{1}{x+i} & 0 \\ \frac{1}{2\ x+2\ i} & -\frac{1}{2} \end{bmatrix}$$

<div align="center">Type: Union(Matrix Fraction Polynomial Complex Integer,...)</div>

0.1.8 Pattern Matching

A convenient facility for symbolic computation is "pattern matching." Suppose you have a trigonometric expression and you want to transform it to some equivalent form. Use a *rule* command to describe the transformation rules you need. Then give the rules a name and apply that name as a function to your trigonometric expression.

Introduce two rewrite rules.

```
sinCosExpandRules := rule
  sin(x+y) == sin(x)*cos(y) + sin(y)*cos(x)
  cos(x+y) == cos(x)*cos(y) - sin(x)*sin(y)
  sin(2*x) == 2*sin(x)*cos(x)
  cos(2*x) == cos(x)**2 - sin(x)**2


  {sin(y + x) == cos(x)sin(y) + cos(y)sin(x),
   cos(y + x) == - sin(x)sin(y) + cos(x)cos(y),
   sin(2x) == 2cos(x)sin(x),
                     2           2
   cos(2x) == - sin(x)  + cos(x) }
```

<div align="center">Type: Ruleset(Integer,Integer,Expression Integer)</div>

Apply the rules to a simple trigonometric expression.

```
sinCosExpandRules(sin(a+2*b+c))
```

$$\left(-\cos\left(a\right)\sin\left(b\right)^2 - 2\,\cos\left(b\right)\sin\left(a\right)\sin\left(b\right) + \cos\left(a\right)\cos\left(b\right)^2\right)\sin\left(c\right) -$$

$$\cos\left(c\right)\sin\left(a\right)\sin\left(b\right)^2 + 2\,\cos\left(a\right)\cos\left(b\right)\cos\left(c\right)\sin\left(b\right) +$$

$$\cos\left(b\right)^2\cos\left(c\right)\sin\left(a\right)$$

<div align="right">Type: Expression Integer</div>

Using input files, you can create your own library of transformation rules relevant to your applications, then selectively apply the rules you need.

0.1.9   Polymorphic Algorithms

All components of the Axiom algebra library are written in the Axiom library language. This language is similar to the interactive language except for protocols that authors are obliged to follow. The library language permits you to write "polymorphic algorithms," algorithms defined to work in their most natural settings and over a variety of types.

Define a system of polynomial equations S.

```
S := [3*x**3 + y + 1 = 0,y**2 = 4]
```

$$\left[y + 3\,x^3 + 1 = 0, y^2 = 4\right]$$

<div align="right">Type: List Equation Polynomial Integer</div>

Solve the system S using rational number arithmetic and 30 digits of accuracy.

```
solve(S,1/10**30)
```

$$\left[\left[y = -2, x = \frac{1757879671211184245283070414507}{2535301200456458802993406410752}\right], \left[y = 2, x = -1\right]\right]$$

<div align="right">Type: List List Equation Polynomial Fraction Integer</div>

Solve S with the solutions expressed in radicals.

```
radicalSolve(S)
```

$$\left[[y = 2, x = -1], \left[y = 2, x = \frac{-\sqrt{-3} + 1}{2} \right], \right.$$

$$\left[y = 2, x = \frac{\sqrt{-3} + 1}{2} \right], \left[y = -2, x = \frac{1}{\sqrt[3]{3}} \right],$$

$$\left. \left[y = -2, x = \frac{\sqrt{-1}\,\sqrt{3} - 1}{2\,\sqrt[3]{3}} \right], \left[y = -2, x = \frac{-\sqrt{-1}\,\sqrt{3} - 1}{2\,\sqrt[3]{3}} \right] \right]$$

                    Type: List List Equation Expression Integer

While these solutions look very different, the results were produced by the same internal algorithm! The internal algorithm actually works with equations over any "field." Examples of fields are the rational numbers, floating point numbers, rational functions, power series, and general expressions involving radicals.

0.1.10   Extensibility

Users and system developers alike can augment the Axiom library, all using one common language. Library code, like interpreter code, is compiled into machine binary code for run-time efficiency.

Using this language, you can create new computational types and new algorithmic packages. All library code is polymorphic, described in terms of a database of algebraic properties. By following the language protocols, there is an automatic, guaranteed interaction between your code and that of colleagues and system implementers.

A Technical Introduction

Axiom has both an *interactive language* for user interactions and a *programming language* for building library modules. Like Modula 2, PASCAL, FORTRAN, and Ada, the programming language emphasizes strict type-checking. Unlike these languages, types in Axiom are dynamic objects: they are created at runtime in response to user commands.

Here is the idea of the Axiom programming language in a nutshell. Axiom types range from algebraic ones (like polynomials, matrices, and power series) to data structures (like lists, dictionaries, and input files). Types combine in any meaningful way. You can build polynomials of matrices, matrices of polynomials of power series, hash tables with symbolic keys and rational function entries, and so on.

Categories define algebraic properties to ensure mathematical correctness. They ensure, for example, that matrices of polynomials are OK, but matrices of input files are not. Through categories, programs can discover that polynomials of continued fractions have a commutative multiplication whereas polynomials of matrices do not.

Categories allow algorithms to be defined in their most natural setting. For example, an algorithm can be defined to solve polynomial equations over *any* field. Likewise a greatest common divisor can compute the "gcd" of two elements from *any* Euclidean domain. Categories foil attempts to compute meaningless "gcds", for example, of two hashtables. Categories also enable algorithms to be compiled into machine code that can be run with arbitrary types.

The Axiom interactive language is oriented towards ease-of-use. The Axiom interpreter uses type-inferencing to deduce the type of an object from user input. Type declarations can generally be omitted for common types in the interactive language.

So much for the nutshell. Here are these basic ideas described by ten design principles:

0.1.11 Types are Defined by Abstract Datatype Programs

Basic types are called *domains of computation*, or, simply, *domains.* Domains are defined by Axiom programs of the form:

```
Name(...): Exports == Implementation
```

Each domain has a capitalized `Name` that is used to refer to the class of its members. For example, `Integer` denotes "the class of integers," `Float`, "the class of floating point numbers," and `String`, "the class of strings."

The "..." part following `Name` lists zero or more parameters to the constructor. Some basic ones like `Integer` take no parameters. Others, like `Matrix`, `Polynomial` and `List`, take a single parameter that again must be a domain. For example, `Matrix(Integer)` denotes "matrices over the integers," `Polynomial (Float)` denotes "polynomial with floating point coefficients," and `List (Matrix (Polynomial (Integer)))` denotes "lists of matrices of polynomials over the integers." There is no restriction on the number or type of parameters of a domain constructor.

SquareMatrix(2,Integer) is an example of a domain constructor that accepts both a particular data value as well as an integer. In this case the number 2 specifies the number of rows and columns the square matrix will contain. Elements of the matricies are integers.

The `Exports` part specifies operations for creating and manipulating objects of the domain. For example, type `Integer` exports constants 0 and 1, and operations "+", "−", and "∗". While these operations are common, others such as **odd?** and **bit?** are not. In addition the Exports section can contain symbols that represent properties that can be tested. For example, the Category `EntireRing` has the symbol `noZeroDivisors` which asserts that if a product is zero then one of the factors must be zero.

The `Implementation` part defines functions that implement the exported operations of the domain. These functions are frequently described in terms of another lower-level domain used to represent the objects of the domain. Thus the operation of adding two vectors of real numbers can be described and implemented using the addition operation from `Float`.

0.1.12   The Type of Basic Objects is a Domain or Subdomain

Every Axiom object belongs to a *unique* domain. The domain of an object is also called its *type*. Thus the integer 7 has type `Integer` and the string `"daniel"` has type `String`.

The type of an object, however, is not unique. The type of integer 7 is not only `Integer` but `NonNegativeInteger`, `PositiveInteger`, and possibly, in general, any other "subdomain" of the domain `Integer`. A *subdomain* is a domain with a "membership predicate". `PositiveInteger` is a subdomain of `Integer` with the predicate "is the integer > 0?".

Subdomains with names are defined by abstract datatype programs similar to those for domains. The *Export* part of a subdomain, however, must list a subset of the exports of the domain. The `Implementation` part optionally gives special definitions for subdomain objects.

0.1.13 Domains Have Types Called Categories

Domain and subdomains in Axiom are themselves objects that have types. The type of a domain or subdomain is called a *category*. Categories are described by programs of the form:

```
Name(...): Category == Exports
```

The type of every category is the distinguished symbol `Category`. The category `Name` is used to designate the class of domains of that type. For example, category `Ring` designates the class of all rings. Like domains, categories can take zero or more parameters as indicated by the "`...`" part following `Name`. Two examples are `Module(R)` and `MatrixCategory(R,Row,Col)`.

The `Exports` part defines a set of operations. For example, `Ring` exports the operations "0", "1", "+", "−", and "*". Many algebraic domains such as `Integer` and `Polynomial (Float)` are rings. `String` and `List (R)` (for any domain R) are not.

Categories serve to ensure the type-correctness. The definition of matrices states `Matrix(R: Ring)` requiring its single parameter R to be a ring. Thus a "matrix of polynomials" is allowed, but "matrix of lists" is not.

Categories say nothing about representation. Domains, which are instances of category types, specify representations.

0.1.14 Operations Can Refer To Abstract Types

All operations have prescribed source and target types. Types can be denoted by symbols that stand for domains, called "symbolic domains." The following lines of Axiom code use a symbolic domain R:

```
R: Ring
power: (R, NonNegativeInteger): R -> R
power(x, n) == x ** n
```

Line 1 declares the symbol R to be a ring. Line 2 declares the type of *power* in terms of R. From the definition on line 3, $power(3,2)$ produces 9 for $x = 3$ and $R = $ `Integer`. Also, $power(3.0, 2)$ produces 9.0 for $x = 3.0$ and $R = $ `Float`. $power("oxford", 2)$ however fails since $"oxford"$ has type `String` which is not a ring.

Using symbolic domains, algorithms can be defined in their most natural or general setting.

0.1.15   Categories Form Hierarchies

Categories form hierarchies (technically, directed-acyclic graphs). A simplified hierarchical world of algebraic categories is shown below. At the top of this world is `SetCategory`, the class of algebraic sets. The notions of parents, ancestors, and descendants is clear. Thus ordered sets (domains of category `OrderedSet`) and rings are also algebraic sets. Likewise, fields and integral domains are rings and algebraic sets. However fields and integral domains are not ordered sets.

```
SetCategory +---- Ring          ---- IntegralDomain ---- Field
            |
            +---- Finite      ---+
            |                     \
            +---- OrderedSet -----+ OrderedFinite
```
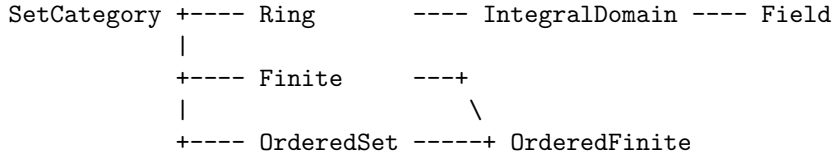
Figure 1. A simplified category hierarchy.

0.1.16   Domains Belong to Categories by Assertion

A category designates a class of domains. Which domains? You might think that `Ring` designates the class of all domains that export 0, 1, "+", "-", and "*". But this is not so. Each domain must *assert* which categories it belongs to.

The `Export` part of the definition for `Integer` reads, for example:

```
Join(OrderedSet, IntegralDomain,  ...) with ...
```

This definition asserts that `Integer` is both an ordered set and an integral domain. In fact, `Integer` does not explicitly export constants 0 and 1 and operations "+", "-" and "*" at all: it inherits them all from *Ring*! Since `IntegralDomain` is a descendant of *Ring*, `Integer` is therefore also a ring.

Assertions can be conditional. For example, `Complex(R)` defines its exports by:

```
Ring with ... if R has Field then Field ...
```

Thus `Complex(Float)` is a field but `Complex(Integer)` is not since `Integer` is not a field.

You may wonder: "Why not simply let the set of operations determine whether a domain belongs to a given category?". Axiom allows operation names (for example, **norm**) to have very different meanings in different contexts. The meaning of an operation in Axiom is determined by context. By associating operations with categories, operation names can be reused whenever appropriate or convenient to do so. As a simple example, the operation < might be used to

denote lexicographic-comparison in an algorithm. However, it is wrong to use the same `<` with this definition of absolute-value:

$$abs(x) == if\ x < 0\ then - x\ else\ x$$

Such a definition for `abs` in Axiom is protected by context: argument x is required to be a member of a domain of category `OrderedSet`.

0.1.17 Packages Are Clusters of Polymorphic Operations

In Axiom, facilities for symbolic integration, solution of equations, and the like are placed in "packages". A *package* is a special kind of domain: one whose exported operations depend solely on the parameters of the constructor and/or explicit domains. Packages, unlike Domains, do not specify the representation.

If you want to use Axiom, for example, to define some algorithms for solving equations of polynomials over an arbitrary field F, you can do so with a package of the form:

```
MySolve(F: Field): Exports == Implementation
```

where `Exports` specifies the **solve** operations you wish to export from the domain and the `Implementation` defines functions for implementing your algorithms. Once Axiom has compiled your package, your algorithms can then be used for any `F`: floating-point numbers, rational numbers, complex rational functions, and power series, to name a few.

0.1.18 The Interpreter Builds Domains Dynamically

The Axiom interpreter reads user input then builds whatever types it needs to perform the indicated computations. For example, to create the matrix

$$M = \begin{pmatrix} x^2 + 1 & 0 \\ 0 & x/2 \end{pmatrix}$$

using the command:

```
M = [ [x**2+1,0],[0,x / 2] ]::Matrix(POLY(FRAC(INT)))
```

$$M = \begin{bmatrix} x^2 + 1 & 0 \\ 0 & x/2 \end{bmatrix}$$

                Type: Matrix Polynomial Fraction Integer

the interpreter first loads the modules `Matrix`, `Polynomial`, `Fraction`, and `Integer` from the library, then builds the *domain tower* "matrices of polynomials of rational numbers (i.e. fractions of integers)".

You can watch the loading process by first typing

```
)set message autoload on
```

In addition to the named domains above many additional domains and categories are loaded. Most systems are preloaded with such common types. For efficiency reasons the most common domains are preloaded but most (there are more than 1100 domains, categories, and packages) are not. Once these domains are loaded they are immediately available to the interpreter.

Once a domain tower is built, it contains all the operations specific to the type. Computation proceeds by calling operations that exist in the tower. For example, suppose that the user asks to square the above matrix. To do this, the function "`*`" from `Matrix` is passed the matrix M to compute $M * M$. The function is also passed an environment containing R that, in this case, is `Polynomial (Fraction (Integer))`. This results in the successive calling of the "`*`" operations from `Polynomial`, then from `Fraction`, and then finally from `Integer`.

Categories play a policing role in the building of domains. Because the argument of `Matrix` is required to be a `Ring`, Axiom will not build nonsensical types such as "matrices of input files".

0.1.19 Axiom Code is Compiled

Axiom programs are statically compiled to machine code, then placed into library modules. Categories provide an important role in obtaining efficient object code by enabling:

- static type-checking at compile time;

- fast linkage to operations in domain-valued parameters;

- optimization techniques to be used for partially specified types (operations for "vectors of R", for instance, can be open-coded even though `R` is unknown).

0.1.20 Axiom is Extensible

Users and system implementers alike use the Axiom language to add facilities to the Axiom library. The entire Axiom library is in fact written in the Axiom source code and available for user modification and/or extension.

Axiom's use of abstract datatypes clearly separates the exports of a domain (what operations are defined) from its implementation (how the objects are represented and operations are defined). Users of a domain can thus only create and manipulate objects through these exported operations. This allows implementers to "remove and replace" parts of the library safely by newly upgraded (and, we hope, correct) implementations without consequence to its users.

Categories protect names by context, making the same names available for use in other contexts. Categories also provide for code-economy. Algorithms can be parameterized categorically to characterize their correct and most general context. Once compiled, the same machine code is applicable in all such contexts.

Finally, Axiom provides an automatic, guaranteed interaction between new and old code. For example:

- if you write a new algorithm that requires a parameter to be a field, then your algorithm will work automatically with every field defined in the system; past, present, or future.

- if you introduce a new domain constructor that produces a field, then the objects of that domain can be used as parameters to any algorithm using field objects defined in the system; past, present, or future.

These are the key ideas. For further information, we particularly recommend your reading chapters 11, 12, and 13, where these ideas are explained in greater detail.

0.2 Using Axiom as a Pocket Calculator

At the simplest level Axiom can be used as a pocket calculator where expressions involving numbers and operators are entered directly in infix notation. In this sense the more advanced features of the calculator can be regarded as operators (e.g **sin**, **cos**, etc).

0.2.1 Basic Arithmetic

An example of this might be to calculate the cosine of 2.45 (in radians). To do this one would type:

```
cos 2.45
```

$$-0.7702312540473073417$$

```
                                             Type:  Float
```

Before proceeding any further it would be best to explain the previous three lines. Firstly the text "(1) `->` " is part of the prompt that the Axiom system provides when in interactive mode. The full prompt has other text preceding this but it is not relevant here. The number in parenthesis is the step number of the input which may be used to refer to the *results* of previous calculations. The step number appears at the start of the second line to tell you which step

the result belongs to. Since the interpreter probably loaded numberous libraries to calculate the result given above and listed each one in the prcess, there could easily be several pages of text between your input and the answer.

The last line contains the type of the result. The type `Float` is used to represent real numbers of arbitrary size and precision (where the user is able to define how big arbitrary is – the default is 20 digits but can be as large as your computer system can handle). The type of the result can help track down mistakes in your input if you don't get the answer you expected.

Other arithmetic operations such as addition, subtraction, and multiplication behave as expected:

```
6.93 * 4.1328
```

$$28.640304$$

                                                                      Type: Float

```
6.93 / 4.1328
```

$$1.6768292682926829268$$

                                                                      Type: Float

but integer division isn't quite so obvious. For example, if one types:

```
4/6
```

$$\frac{2}{3}$$

                                                          Type: Fraction Integer

a fractional result is obtained. The function used to display fractions attempts to produce the most readable answer. In the example:

```
4/2
```

$$2$$

                                                          Type: Fraction Integer

the result is stored as the fraction 2/1 but is displayed as the integer 2. This fraction could be converted to type `Integer` with no loss of informatin but Axiom will not do so automatically.

0.2.2 Type Conversion

To obtain the floating point value of a fraction one must convert (**conversions** are applied by the user and **coercions** are applied automatically by the interpreter) the result to type `Float` using the "::" operator as follows:

`(4.6)::Float`

$$4.6$$

<div align="right">Type: Float</div>

Although Axiom can convert this back to a fraction it might not be the same fraction you started with as due to rounding errors. For example, the following conversion appears to be without error but others might not:

`%::Fraction Integer`

$$\frac{23}{5}$$

<div align="right">Type: Fraction Integer</div>

where "%" represents the previous *result* (not the calculation).

Although Axiom has the ability to work with floating-point numbers to a very high precision it must be remembered that calculations with these numbers are **not** exact. Since Axiom is a computer algebra package and not a numerical solutions package this should not create too many problems. The idea is that the user should use Axiom to do all the necessary symbolic manipulation and only at the end should actual numerical results be extracted.

If you bear in mind that Axiom appears to store expressions just as you have typed them and does not perform any evalutation of them unless forced to then programming in the system will be much easier. It means that anything you ask Axiom to do (within reason) will be carried with complete accuracy.

In the previous examples the "::" operator was used to convert values from one type to another. This type conversion is not possible for all values. For instance, it is not possible to convert the number 3.4 to an integer type since it can't be represented as an integer. The number 4.0 can be converted to an integer type since it has no fractional part.

Conversion from floating point values to integers is performed using the functions **round** and **truncate**. The first of these rounds a floating point number to the nearest integer while the other truncates (i.e. removes the fractional part). Both functions return the result as a **floating point** number. To extract the

fractional part of a floating point number use the function **fractionPart** but
note that the sign of the result depends on the sign of the argument. Axiom
obtains the fractional part of x using $x - truncate(x)$:

round(3.77623)

$$4.0$$

Type: Float

round(-3.77623)

$$-4.0$$

Type: Float

truncate(9.235)

$$9.0$$

Type: Float

truncate(-9.654)

$$-9.0$$

Type: Float

fractionPart(-3.77623)

$$-0.77623$$

Type: Float

0.2.3 Useful Functions

To obtain the absolute value of a number the **abs** function can be used. This takes a single argument which is usually an integer or a floating point value but doesn't necessarily have to be. The sign of a value can be obtained via the **sign** function which rturns -1, 0, or 1 depending on the sign of the argument.

```
abs(4)
```

$$4$$

Type: PositiveInteger

```
abs(-3)
```

$$3$$

Type: PositiveInteger

```
abs(-34254.12314)
```

$$34254.12314$$

Type: Float

```
sign(-49543.2345346)
```

$$-1$$

Type: Integer

```
sign(0)
```

$$0$$

Type: NonNegativeInteger

```
sign(234235.42354)
```

$$1$$

```
                                              Type:   PositiveInteger
```

Tests on values can be done using various functions which are generally more efficient than using relational operators such as = particularly if the value is a matrix. Examples of some of these functions are:

```
positive?(-234)
```

```
                                        false
```

```
                                              Type:   Boolean
```

```
negative?(-234)
```

```
                                        true
```

```
                                              Type:   Boolean
```

```
zero?(42)
```

```
                                        false
```

```
                                              Type:   Boolean
```

```
one?(1)
```

```
                                        true
```

```
                                              Type:   Boolean
```

```
odd?(23)
```

```
                                        true
```

```
                                              Type:   Boolean
```

```
odd?(9.435)
```

```
                                        false
```

                                                    Type: Boolean

even?(-42)

                                 true

                                                    Type: Boolean

prime?(37)

                                 true

                                                    Type: Boolean

prime?(-37)

                                 false

                                                    Type: Boolean

Some other functions that are quite useful for manipulating numerical values
are:

```
sin(x)        Sine of x
cos(x)        Cosine of x
tan(x)        Tangent of x
asin(x)       Arcsin of x
acos(x)       Arccos of x
atan(x)       Arctangent of x
gcd(x,y)      Greatest common divisor of x and y
lcm(x,y)      Lowest common multiple of x and y
max(x,y)      Maximum of x and y
min(x,y)      Minimum of x and y
factorial(x)  Factorial of x
factor(x)     Prime factors of x
divide(x,y)   Quotient and remainder of x/y
```

Some simple infix and prefix operators:

```
+     Addition              -     Subtraction
-     Numerical Negation    ~     Logical Negation
/\    Conjunction (AND)     \/    Disjunction (OR)
```

```
and     Logical AND (/\)      or      Logical OR (\/)
not     Logical Negation      **      Exponentiation
*       Multiplication        /       Division
quo     Quotient              rem     Remainder
<       less than             >       greater than
<=      less than or equal    >=      greater than or equal
```

Some useful Axiom macros:

```
%i              The square root of -1
%e              The base of the natural logarithm
%pi             Pi
%infinity       Infinity
%plusInfinity   Positive Infinity
%minusInfinity  Negative Infinity
```

0.3 Using Axiom as a Symbolic Calculator

In the previous section all the examples involved numbers and simple functions. Also none of the expressions entered were assigned to anything. In this section we will move on to simple algebra (i.e. expressions involving symbols and other features available on more sophisticated calculators).

0.3.1 Expressions Involving Symbols

Expressions involving symbols are entered just as they are written down, for example:

```
xSquared := x**2
```

$$x^2$$

Type: Polynomial Integer

where the assignment operator ":=" represents immediate assignment. Later it will be seen that this form of assignment is not always desirable and the use of the delayed assignment operator "==" will be introduced. The type of the result is `Polynomial Integer` which is used to represent polynomials with integer coefficients. Some other examples along similar lines are:

```
xDummy := 3.21*x**2
```

$$3.21 \ x^2$$

Type: Polynomial Float

```
xDummy := x**2.5
```

$$x^2 \sqrt{x}$$

Type: Expression Float

```
xDummy := x**3.3
```

$$x^3 \sqrt[10]{x^3}$$

Type: Expression Float

```
xyDummy := x**2 - y**2
```

$$-y^2 + x^2$$

Type: Polynomial Integer

Given that we can define expressions involving symbols, how do we actually compute the result when the symbols are assigned values? The answer is to use the **eval** function which takes an expression as its first argument followed by a list of assignments. For example, to evaluate the expressions **XDummy** and xyDummy resulting from their respective assignments above we type:

```
eval(xDummy,x=3)
```

$$37.540507598529552193$$

Type: Expression Float

```
eval(xyDummy, [x=3, y=2.1])
```

$$4.59$$

Type: Polynomial Float

0.3.2   Complex Numbers

For many scientific calculations real numbers aren't sufficient and support for complex numbers is also required. Complex numbers are handled in an intuitive manner and Axiom, which uses the **%i** macro to represent the square root of -1. Thus expressions involving complex numbers are entered just like other expressions.

```
(2/3 + %i)**3
```

$$-\frac{46}{27} + \frac{1}{3}\ i$$

                              Type: Complex Fraction Integer

The real and imaginary parts of a complex number can be extracted using the **real** and **imag** functions and the complex conjugate of a number can be obtained using **conjugate**:

```
real(3 + 2*%i)
```

$$3$$

                                       Type: PositiveInteger

```
imag(3+ 2*%i)
```

$$2$$

                                       Type: PositiveInteger

```
conjugate(3 + 2*%i)
```

$$3 - 2\ i$$

                                       Type: Complex Integer

The function **factor** can also be applied to complex numbers but the results aren't quite so obvious as for factoring integer:

```
144 + 24*%i
```

$$144 + 24\ i$$

                                       Type: Complex Integer

0.3.3 Number Representations

By default all numerical results are displayed in decimal with real numbers shown to 20 significant figures. If the integer part of a number is longer than 20 digits then nothing after the decimal point is shown and the integer part is given in full. To alter the number of digits shown the function **digits** can be called. The result returned by this function is the previous setting. For example, to find the value of π to 40 digits we type:

```
digits(40)
```

$$20$$

<div align="right">Type: PositiveInteger</div>

```
%pi::Float
```

$$3.1415926535\ 8979323846\ 2643383279\ 502884197$$

<div align="right">Type: Float</div>

As can be seen in the example above, there is a gap after every ten digits. This can be changed using the **outputSpacing** function where the argument is the number of digits to be displayed before a space is inserted. If no spaces are desired then use the value 0. Two other functions controlling the appearance of real numbers are **outputFloating** and **outputFixed**. The former causes Axiom to display floating-point values in exponent notation and the latter causes it to use fixed-point notation. For example:

```
outputFloating(); %
```

$$0.3141592653589793238462643383279502884197E1$$

<div align="right">Type: Float</div>

```
outputFloating(3); 0.00345
```

$$0.345E-2$$

<div align="right">Type: Float</div>

```
outputFixed(); %
```

$$0.00345$$

<div align="right">Type: Float</div>

```
outputFixed(3); %
```

$$0.003$$

<div align="right">Type: Float</div>

```
outputGeneral(); %
```

$$0.00345$$

<div align="right">Type: Float</div>

Note that the semicolon ";" in the examples above allows several expressions to be entered on one line. The result of the last expression is displayed. remember also that the percent symbol "%" is used to represent the result of a previous calculation.

To display rational numbers in a base other than 10 the function **radix** is used. The first argument of this function is the expression to be displayed and the second is the base to be used.

```
radix(10**10,32)
```

$$9A0NP00$$

<div align="right">Type: RadixExpansion 32</div>

```
radix(3/21,5)
```

$$0.\overline{032412}$$

<div align="right">Type: RadixExpansion 5</div>

Rational numbers can be represented as a repeated decimal expansion using the **decimal** function or as a continued fraction using **continuedFraction**. Any attempt to call these functions with irrational values will fail.

```
decimal(22/7)
```

$$3.\overline{142857}$$

Type: DecimalExpansion

continuedFraction(6543/210)

$$31 + \frac{1|}{|6} + \frac{1|}{|2} + \frac{1|}{|1} + \frac{1|}{|3}$$

Type: ContinuedFraction Integer

Finally, partial fractions in compact and expanded form are available via the functions **partialFraction** and **padicFraction** respectively. The former takes two arguments, the first being the numerator of the fraction and the second being the denominator. The latter function takes a fraction and expands it further while the function **compactFraction** does the reverse:

partialFraction(234,40)

$$6 - \frac{3}{2^2} + \frac{3}{5}$$

Type: PartialFraction Integer

padicFraction(%)

$$6 - \frac{1}{2} - \frac{1}{2^2} + \frac{3}{5}$$

Type: PartialFraction Integer

compactFraction(%)

$$6 - \frac{3}{2^2} + \frac{3}{5}$$

Type: PartialFraction Integer

padicFraction(234/40)

$$\frac{117}{20}$$

To extract parts of a partial fraction the function **nthFractionalTerm** is available and returns a partial fraction of one term. To decompose this further the numerator can be obtained using **firstNumer** and the denominator with **firstDenom**. The whole part of a partial fraction can be retrieved using **wholePart** and the number of fractional parts can be found using the function **numberOf FractionalTerms**:

```
t := partialFraction(234,40)
```

$$6 - \frac{3}{2^2} + \frac{3}{5}$$

                                        Type: PartialFraction Integer

```
wholePart(t)
```

$$6$$

                                                Type: PositiveInteger

```
numberOfFractionalTerms(t)
```

$$2$$

                                                Type: PositiveInteger

```
p := nthFractionalTerm(t,1)
```

$$-\frac{3}{2^2}$$

                                            Type: PartialFraction Integer

```
firstNumer(p)
```

$$-3$$

                                                    Type: Integer

```
firstDenom(p)
```

$$2^2$$

                                                Type: Factored Integer

0.3.4   Modular Arithmetic

By using the type constructor `PrimeField` it is possible to do arithmetic modulo
some prime number. For example, arithmetic module 7 can be performed as
follows:

```
x :  PrimeField 7 := 5
```

$$5$$

<div align="right">Type: PrimeField 7</div>

```
x**5 + 6
```

$$2$$

<div align="right">Type: PrimeField 7</div>

```
1/x
```

$$3$$

<div align="right">Type: PrimeField 7</div>

The first example should be read as:

>   Let x be of type `PrimeField(7)` and assign to it the value 5

Note that it is only possible to invert non-zero values if the arithmetic is per-
formed modulo a prime number. Thus arithmetic modulo a non-prime integer
is possible but the reciprocal operation is undefined and will generate an error.
Attempting to use the `PrimeField` type constructor with a non-prime argument
will generate an error. An example of non-prime modulo arithmetic is:

```
y :  IntegerMod 8 := 11
```

$$3$$

<div align="right">Type: IntegerMod 8</div>

```
y*4 + 27
```

7

Note that polynomials can be constructed in a similar way:

```
(3*a**4 + 27*a - 36)::Polynomial PrimeField 7
```

$$3\ a^4 + 6\ a + 6$$

                                                     Type: Polynomial PrimeField 7

0.4 General Points about Axiom

0.4.1 Computation Without Output

It is sometimes desirable to enter an expression and prevent Axiom from displaying the result. To do this the expression should be terminated with a semicolon ";". In a previous section it was mentioned that a set of expressions separated by semicolons would be evaluated and the result of the last one displayed. Thus if a single expression is followed by a semicolon no output will be produced (except for its type):

```
2 + 4*5;
```

                                                          Type: PositiveInteger

0.4.2 Accessing Earlier Results

The "%" macro represents the result of the previous computation. The "%%" macro is available which takes a single integer argument. If the argument is positive then it refers to the step number of the calculation where the numbering begins from one and can be seen at the end of each prompt (the number in parentheses). If the argument is negative then it refers to previous results counting backwards from the last result. That is, "%%(-1)" is the same as "%". The value of "%%(0)" is not defined and will generate an error if requested.

0.4.3   Splitting Expressions Over Several Lines

Although Axiom will quite happily accept expressions that are longer than the width of the screen (just keep typing without pressing the **Return** key) it is often preferable to split the expression being entered at a point where it would result in more readable input. To do this the underscore "_" symbol is placed before the break point and then the **Return** key is pressed. The rest of the expression is typed on the next line, can be preceeded by any number of whitespace chars, for example:

```
2_
+_
3
```

$$5$$

<p align="right">Type: PositiveInteger</p>

The underscore symbol is an excape character and its presence alters the meaning of the characters that follow it. As mentions above whitespace following an underscore is ignored (the **Return** key generates a whitespace character). Any other character following an underscore loses whatever special meaning it may have had. Thus one can create the identifier "a+b" by typing "a_+b" although this might lead to confusions. Also note the result of the following example:

```
ThisIsAVeryLong_
VariableName
```

$$ThisIsAVeryLongVariableName$$

<p align="right">Type: Variable ThisIsAVeryLongVariableName</p>

0.4.4   Comments and Descriptions

Comments and descriptions are really only of use in files of Axiom code but can be used when the output of an interactive session is being spooled to a file (via the system command **)spool**). A comment begins with two dashes "- -" and continues until the end of the line. Multi-line comments are only possible if each individual line begins with two dashes.

Descriptions are the same as comments except that the Axiom compiler will include them in the object files produced and make them availabe to the end user for documentation purposes.

A description is placed **before** a calculation begins with three "+++" signs and a description placed after a calculation begins with two plus symbols "+".

The so-called "plus plus" comments are used within the algebra files and are processed by the compiler to add to the documentation. The so-called "minus minus" comments are ignored everywhere.

0.4.5   Control of Result Types

In earlier sections the type of an expression was converted to another via the "::" operator. However, this is not the only method for converting between types and two other operators need to be introduced and explained.

The first operator is "$" and is used to specify the package to be used to calculate the result. Thus:

```
(2/3)$Float
```

$$0.6666666666\ 6666666667$$

                                                                                       Type: Float

tells Axiom to use the "/" operator from the `Float` package to evaluate the expression 2/3. This doe not necessarily mean that the result will be of the same type as the domain from which the operator was taken. In the following example the **sign** operator is taken from the `Float` package but the result is of type `Integer`.

```
sign(2.3)$Float
```

                                                          1

                                                                                       Type: Integer

The other operator is "@" which is used to tell Axiom what the desired type of the result of the calculation is. In most situations all three operators yield the same results but the example below should help distinguish them.

```
(2 + 3)::String
```

                                                      "5"

                                                                                       Type: String

```
(2 + 3)@String
```

```
An expression involving @ String actually evaluated to one of
   type PositiveInteger . Perhaps you should use :: String .
```

```
(2 + 3)$String
```

```
   The function + is not implemented in String .
```

If an expression X is converted using one of the three operators to type T the interpretations are:

:: means explicitly convert X to type T if possible.

$ means use the available operators for type T to compute X.

@ means choose operators to compute X so that the result is of type T.

0.5   Data Structures in Axiom

This chapter is an overview of *some* of the data structures provided by Axiom.

0.5.1   Lists

The Axiom `List` type constructor is used to create homogenous lists of finite size. The notation for lists and the names of the functions that operate over them are similar to those found in functional languages such as ML.

Lists can be created by placing a comma separated list of values inside square brackets or if a list with just one element is desired then the function **list** is available:

```
[4]
```

$$[4]$$

```
                                        Type:  List PositiveInteger
```

```
list(4)
```

$$[4]$$

```
                                        Type:  List PositiveInteger
```

```
[1,2,3,5,7,11]
```

$$[1, 2, 3, 5, 7, 11]$$

                                      Type: List PositiveInteger

The function **append** takes two lists as arguments and returns the list consisting
of the second argument appended to the first. A single element can be added
to the front of a list using **cons**:

```
append([1,2,3,5],[7,11])
```

$$[1, 2, 3, 5, 7, 11]$$

                                      Type: List PositiveInteger

```
cons(23,[65,42,19])
```

$$[23, 65, 42, 19]$$

                                      Type: List PositiveInteger

Lists are accessed sequentially so if Axiom is asked for the value of the twentieth
element in the list it will move from the start of the list over nineteen elements
before it reaches the desired element. Each element of a list is stored as a node
consisting of the value of the element and a pointer to the rest of the list. As a
result the two main operations on a list are called **first** and **rest**. Both of these
functions take a second optional argument which specifies the length of the first
part of the list:

```
first([1,5,6,2,3])
```

$$1$$

                                      Type: PositiveInteger

```
first([1,5,6,2,3],2)
```

$$[1, 5]$$

                                      Type: List PositiveInteger

```
rest([1,5,6,2,3])
```

$$[5, 6, 2, 3]$$

<div align="right">Type: List PositiveInteger</div>

```
rest([1,5,6,2,3],2)
```

$$[6, 2, 3]$$

<div align="right">Type: List PositiveInteger</div>

Other functions are **empty?** which tests to see if a list contains no elements, **member?** which tests to see if the first argument is a member of the second, **reverse** which reverses the order of the list, **sort** which sorts a list, and **removeDuplicates** which removes any duplicates. The length of a list can be obtained using the "#" operator.

```
empty?([7,2,-1,2])
```

<div align="center">false</div>

<div align="right">Type: Boolean</div>

```
member?(-1,[7,2,-1,2])
```

<div align="center">true</div>

<div align="right">Type: Boolean</div>

```
reverse([7,2,-1,2])
```

$$[2, -1, 2, 7]$$

<div align="right">Type: List Integer</div>

```
sort([7,2,-1,2])
```

$$[-1, 2, 2, 7]$$

<div align="right">Type: List Integer</div>

```
removeDuplicates([1,5,3,5,1,1,2])
```

$$[1, 5, 3, 2]$$

<div align="right">Type: List PositiveInteger</div>

```
#[7,2,-1,2]
```

<div align="center">4</div>

<div align="right">Type: PositiveInteger</div>

Lists in Axiom are mutable and so their contents (the elements and the links) can be modified in place. Functions that operator over lists in this way have names ending in the symbol "!". For example, **concat!** takes two lists as arguments and appends the second argument to the first (except when the first argument is an empty list) and **setrest!** changes the link emanating from the first argument to point to the second argument:

```
u := [9,2,4,7]
```

$$[9, 2, 4, 7]$$

<div align="right">Type: List PositiveInteger</div>

```
concat!(u,[1,5,42]); u
```

$$[9, 2, 4, 7, 1, 5, 42]$$

<div align="right">Type: List PositiveInteger</div>

```
endOfu := rest(u,4)
```

$$[1, 5, 42]$$

<div align="right">Type: List PositiveInteger</div>

```
partOfu := rest(u,2)
```

$$[4, 7, 1, 5, 42]$$

<div align="right">Type: List PositiveInteger</div>

`setrest!(endOfu,partOfu); u`

$$\left[9, 2, \overline{4, 7, 1}\right]$$

<div align="right">Type: List PositiveInteger</div>

From this it can be seen that the lists returned by **first** and **rest** are pointers to the original list and *not* a copy. Thus great care must be taken when dealing with lists in Axiom.

Although the *n*th element of the list *l* can be obtained by applying the **first** function to $n - 1$ applications of **rest** to *l*, Axiom provides a more useful access method in the form of the "." operator:

`u.3`

<div align="center">4</div>

<div align="right">Type: PositiveInteger</div>

`u.5`

<div align="center">1</div>

<div align="right">Type: PositiveInteger</div>

`u.6`

<div align="center">4</div>

<div align="right">Type: PositiveInteger</div>

`first rest rest u -- Same as u.3`

<div align="center">4</div>

<div align="right">Type: PositiveInteger</div>

`u.first`

$$9$$

<div align="right">Type: PositiveInteger</div>

```
u(3)
```

$$4$$

<div align="right">Type: PositiveInteger</div>

The operation *u.i* is referred to as *indexing into u* or *elting into u*. The latter term comes from the **elt** function which is used to extract elements (the first element of the list is at index 1).

```
elt(u,4)
```

$$7$$

<div align="right">Type: PositiveInteger</div>

If a list has no cycles then any attempt to access an element beyond the end of the list will generate an error. However, in the example above there was a cycle starting at the third element so the access to the sixth element wrapped around to give the third element. Since lists are mutable it is possible to modify elements directly:

```
u.3 := 42; u
```

$$\left[9, 2, \overline{42, 7, 1}\right]$$

<div align="right">Type: List PositiveInteger</div>

Other list operations are:

```
L := [9,3,4,7]; #L
```

$$4$$

<div align="right">Type: PositiveInteger</div>

```
last(L)
```

$$7$$

<div align="right">Type: PositiveInteger</div>

`L.last`

$$7$$

<div align="right">Type: PositiveInteger</div>

`L.(#L - 1)`

$$4$$

<div align="right">Type: PositiveInteger</div>

Note that using the "#" operator on a list with cycles causes Axiom to enter an infinite loop.

Note that any operation on a list L that returns a list LL' will, in general, be such that any changes to LL' will have the side-effect of altering L. For example:

`m := rest(L,2)`

$$[4, 7]$$

<div align="right">Type: List PositiveInteger</div>

`m.1 := 20; L`

$$[9, 3, 20, 7]$$

<div align="right">Type: List PositiveInteger</div>

`n := L`

$$[9, 3, 20, 7]$$

<div align="right">Type: List PositiveInteger</div>

`n.2 := 99; L`

$$[9, 99, 20, 7]$$

                                        Type: List PositiveInteger

n

$$[9, 99, 20, 7]$$

                                        Type: List PositiveInteger

Thus the only save way of copying lists is to copy each element from one to another and not use the assignment operator:

```
p := [i for i in n] -- Same as 'p := copy(n)'
```

$$[9, 99, 20, 7]$$

                                        Type: List PositiveInteger

```
p.2 := 5; p
```

$$[9, 5, 20, 7]$$

                                        Type: List PositiveInteger

n

$$[9, 99, 20, 7]$$

                                        Type: List PositiveInteger

In the previous example a new way of constructing lists was given. This is a powerful method which gives the reader more information about the contents of the list than before and which is extremely flexible. The example

```
[i for i in 1..10]
```

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

                                        Type: List PositiveInteger

should be read as

"Using the expression *i*, generate each element of the list by iterating the
symbol *i* over the range of integers [1,10]"

To generate the list of the squares of the first ten elements we just use:

```
[i**2 for i in 1..10]
```

$$[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]$$

                        Type: List PositiveInteger

For more complex lists we can apply a condition to the elements that are to be
placed into the list to obtain a list of even numbers between 0 and 11:

```
[i for i in 1..10 | even?(i)]
```

$$[2, 4, 6, 8, 10]$$

                        Type: List PositiveInteger

This example should be read as:

"Using the expression *i*, generate each element of the list by iterating the
symbol *i* over the range of integers [1,10] such that *i* is even"

The following achieves the same result:

```
[i for i in 2..10 by 2]
```

$$[2, 4, 6, 8, 10]$$

                        Type: List PositiveInteger

0.5.2   Segmented Lists

A segmented list is one in which some of the elements are ranges of values. The
expand function converts lists of this type into ordinary lists:

```
[1..10]
```

$$[1..10]$$

<div align="right">Type: List Segment PositiveInteger</div>

```
[1..3,5,6,8..10]
```

$$[1..3, 5..5, 6..6, 8..10]$$

<div align="right">Type: List Segment PositiveInteger</div>

```
expand(%)
```

$$[1, 2, 3, 5, 6, 8, 9, 10]$$

<div align="right">Type: List Integer</div>

If the upper bound of a segment is omitted then a different type of segmented
list is obtained and expanding it will produce a stream (which will be considered
in the next section):

```
[1..]
```

$$[1..]$$

<div align="right">Type: List UniversalSegment PositiveInteger</div>

```
expand(%)
```

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \ldots]$$

<div align="right">Type: Stream Integer</div>

0.5.3   Streams

Streams are infinite lists which have the ability to calculate the next element
should it be required. For example, a stream of positive integers and a list of
prime numbers can be generated by:

```
[i for i in 1..]
```

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \ldots]$$

                              Type: Stream PositiveInteger

```
[i for i in 1..  | prime?(i)]
```

$$[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, \ldots]$$

                              Type: Stream PositiveInteger

In each case the first few elements of the stream are calculated for display purposes but the rest of the stream remains unevaluated. The value of items in a stream are only calculated when they are needed which gives rise to their alternative name of "lazy lists".

Another method of creating streams is to use the **generate(f,a)** function. This applies its first argument repeatedly onto its second to produce the stream $[a, f(a), f(f(a)), f(f(f(a))) \ldots]$. Given that the function **nextPrime** returns the lowest prime number greater than its argument we can generate a stream of primes as follows:

```
generate(nextPrime,2)$Stream Integer
```

$$[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, \ldots]$$

                              Type: Stream Integer

As a longer example a stream of Fibonacci numbers will be computed. The Fibonacci numbers start at 1 and each following number is the addition of the two numbers that precede it so the Fibonacci sequence is:

$$1, 1, 2, 3, 5, 8, \ldots$$

.

Since the generation of any Fibonacci number only relies on knowing the previous two numbers we can look at the series through a window of two elements. To create the series the window is placed at the start over the values $[1, 1]$ and their sum obtained. The window is now shifted to the right by one position and the sum placed into the empty slot of the window; the process is then repeated. To implement this we require a function that takes a list of two elements (the current view of the window), adds them, and outputs the new window. The result is the function $[a, b]$ `->` $[b, a + b]$:

```
win :  List Integer -> List Integer
```

<div align="right">Type: Void</div>

```
win(x) == [x.2, x.1 + x.2]
```

<div align="right">Type: Void</div>

```
win([1,1])
```

$$[1, 2]$$

<div align="right">Type: List Integer</div>

```
win(%)
```

$$[2, 3]$$

<div align="right">Type: List Integer</div>

Thus it can be seen that repeatedly applying **win** to the *results* of the previous invocation each element of the series is obtained. Clearly **win** is an ideal function to construct streams using the **generate** function:

```
fibs := [generate(win,[1,1])]
```

$$[[1, 1], [1, 2], [2, 3], [3, 5], [5, 8], [8, 13], [13, 21], [21, 34], [34, 55], [55, 89], \ldots]$$

<div align="right">Type: Stream List Integer</div>

This isn't quite what is wanted – we need to extract the first element of each list and place that in our series:

```
fibs := [i.1 for i in [generate(win,[1,1])] ]
```

$$[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots]$$

<div align="right">Type: Stream Integer</div>

Obtaining the 200th Fibonacci number is trivial:

```
fibs.200
```

$$280571172992510140037611932413038677189525$$

<div align="right">Type: PositiveInteger</div>

One other function of interest is **complete** which expands a finite stream derived from an infinite one (and thus was still stored as an infinite stream) to form a finite stream.

0.5.4 Arrays, Vectors, Strings, and Bits

The simplest array data structure is the *one-dimensional array* which can be obtained by applying the **oneDimensionalArray** function to a list:

```
oneDimensionalArray([7,2,5,4,1,9])
```

$$[7, 2, 5, 4, 1, 9]$$

```
                    Type:  OneDimensionalArray PositiveInteger
```

One-dimensional array are homogenous (all elements must have the same type) and mutable (elements can be changed) like lists but unlike lists they are constant in size and have uniform access times (it is just as quick to read the last element of a one-dimensional array as it is to read the first; this is not true for lists).

Since these arrays are mutable all the warnings that apply to lists apply to arrays. That is, it is possible to modify an element in a copy of an array and change the original:

```
x := oneDimensionalArray([7,2,5,4,1,9])
```

$$[7, 2, 5, 4, 1, 9]$$

```
                    Type:  OneDimensionalArray PositiveInteger
```

```
y := x
```

$$[7, 2, 5, 4, 1, 9]$$

```
                    Type:  OneDimensionalArray PositiveInteger
```

```
y.3 := 20 ; x
```

$$[7, 2, 20, 4, 1, 9]$$

```
                    Type:  OneDimensionalArray PositiveInteger
```

Note that because these arrays are of fixed size the **concat!** function cannot be applied to them without generating an error. If arrays of this type are required use the **FlexibleArray** constructor.

One-dimensional arrays can be created using **new** which specifies the size of the array and the initial value for each of the elements. Other operations that can be applied to one-dimensional arrays are **map!** which applies a mapping onto each element, **swap!** which swaps two elements and **copyInto!(a,b,c)** which copies the array b onto a starting at position c.

```
a :  ARRAY1 PositiveInteger := new(10,3)
```

$$[3, 3, 3, 3, 3, 3, 3, 3, 3, 3]$$

                              Type: OneDimensionalArray PositiveInteger

(note that `ARRAY1` is an abbreviation for the type `OneDimensionalArray`.)
Other types based on one-dimensional arrays are `Vector`, `String`, and tt Bits.

```
map!(i +-> i+1,a); a
```

$$[4, 4, 4, 4, 4, 4, 4, 4, 4, 4]$$

                              Type: OneDimensionalArray PositiveInteger

```
b := oneDimensionalArray([2,3,4,5,6])
```

$$[2, 3, 4, 5, 6]$$

                              Type: OneDimensionalArray PositiveInteger

```
swap!(b,2,3); b
```

$$[2, 4, 3, 5, 6]$$

                              Type: OneDimensionalArray PositiveInteger

```
copyInto!(a,b,3)
```

$$[4, 4, 2, 4, 3, 5, 6, 4, 4, 4]$$

                              Type: OneDimensionalArray PositiveInteger

```
a
```

$$[4, 4, 2, 4, 3, 5, 6, 4, 4, 4]$$

                              Type: OneDimensionalArray PositiveInteger

```
vector([1/2,1/3,1/14])
```

$$\left[\frac{1}{2}, \frac{1}{3}, \frac{1}{14}\right]$$

                                    Type: Vector Fraction Integer

```
"Hello, World"
```

                    "Hello, World"

                                            Type: String

```
bits(8,true)
```

                    "11111111"

                                            Type: Bits

A vector is similar to a one-dimensional array except that if its components belong to a ring then arithmetic operations are provided.

0.5.5   Flexible Arrays

Flexible arrays are designed to provide the efficiency of one-dimensional arrays while retaining the flexibility of lists. They are implemented by allocating a fixed block of storage for the array. If the array needs to be expanded then a larger block of storage is allocated and the contents of the old block are copied into the new one.

There are several operations that can be applied to this type, most of which modify the array in place. As a result these functions all have names ending in "!". The **physicalLength** returns the actual length of the array as stored in memory while the **physicalLength!** allows this value to be changed by the user.

```
f :  FARRAY INT := new(6,1)
```

$$[1, 1, 1, 1, 1, 1]$$

                                    Type: FlexibleArray Integer

```
f.1:=4; f.2:=3 ; f.3:=8 ; f.5:=2 ; f
```

$$[4, 3, 8, 1, 2, 1]$$

```
                                            Type:  FlexibleArray Integer
```

```
insert!(42,f,3); f
```

$$[4, 3, 42, 8, 1, 2, 1]$$

```
                                            Type:  FlexibleArray Integer
```

```
insert!(28,f,8); f
```

$$[4, 3, 42, 8, 1, 2, 1, 28]$$

```
                                            Type:  FlexibleArray Integer
```

```
removeDuplicates!(f)
```

$$[4, 3, 42, 8, 1, 2, 28]$$

```
                                            Type:  FlexibleArray Integer
```

```
delete!(f,5)
```

$$[4, 3, 42, 8, 2, 28]$$

```
                                            Type:  FlexibleArray Integer
```

```
g:=f(3..5)
```

$$[42, 8, 2]$$

```
                                            Type:  FlexibleArray Integer
```

```
g.2:=7; f
```

$$[4, 3, 42, 8, 2, 28]$$

```
                                            Type:  FlexibleArray Integer
```

`insert!(g,f,1)`

$$[42, 7, 2, 4, 3, 42, 8, 2, 28]$$

                                    Type: FlexibleArray Integer

`physicalLength(f)`

$$10$$

                                    Type: PositiveInteger

`physicalLength!(f,20)`

$$[42, 7, 2, 4, 3, 42, 8, 2, 28]$$

                                    Type: FlexibleArray Integer

`merge!(sort!(f),sort!(g))`

$$[2, 2, 2, 3, 4, 7, 7, 8, 28, 42, 42, 42]$$

                                    Type: FlexibleArray Integer

`shrinkable(false)$FlexibleArray(Integer)`

                                    true

                                    Type: Boolean

There are several things to point out concerning these examples. First, although flexible arrays are mutable, making copies of these arrays creates separate entities. This can be seen by the fact that the modification of element *b.2* above did not alter *a*. Second, the **merge!** function can take an extra argument before the two arrays are merged. The argument is a comparison function and defaults to "`<=`" if omitted. Lastly, **shrinkable** tells the system whether or not to let flexible arrays contract when elements are deleted from them. An explicit package reference must be given as in the example above.

0.6   Functions, Choices, and Loops

By now the reader should be able to construct simple one-line expressions involving variables and different data structures. This section builds on this knowledge and shows how to use iteration, make choices, and build functions in Axiom. At the moment it is assumed that the reader has a rough idea of how types are specified and constructed so that they can follow the examples given.

From this point on most examples will be taken from input files.

0.6.1   Reading Code from a File

Input files contain code that will be fed to the command prompt. The primary different between the command line and an input file is that indentation matters. In an input file you can specify "piles" of code by using indentation.

The names of all input files in Axiom should end in ".input" otherwise Axiom will refuse to read them.

If an input file is named **foo.input** you can feed the contents of the file to the command prompt (as though you typed them) by writing: **)read foo.input**.

It is good practice to start each input file with the **)clear all** command so that all functions and variables in the current environment are erased.

0.6.2   Blocks

The Axiom constructs that provide looping, choices, and user-defined functions all rely on the notion of blocks. A block is a sequence of expressions which are evaluated in the order that they appear except when it is modified by control expressions such as loops. To leave a block prematurely use an expression of the form: *BoolExpr => Expr* where *BoolExpr* is any Axiom expression that has type `Boolean`. The value and type of *Expr* determines the value and type returned by the block.

If blocks are entered at the keyboard (as opposed to reading them from a text file) then there is only one way of creating them. The syntax is:

$$(expression1; expression2; \ldots; expressionN)$$

In an input file a block can be constructed as above or by placing all the statements at the same indentation level. When indentation is used to indicate program structure the block is called a *pile*. As an example of a simple block a list of three integers can be constructed using parentheses:

```
( a:=4; b:=1; c:=9; L:=[a,b,c])
```

$$[4, 1, 9]$$

```
                                    Type:  List PositiveInteger
```

Doing the same thing using piles in an input file you could type:

```
L :=
  a:=4
  b:=1
  c:=9
  [a,b,c]
```

$$[4, 1, 9]$$

```
                                    Type:  List PositiveInteger
```

Since blocks have a type and a value they can be used as arguments to functions or as part of other expressions. It should be pointed out that the following example is not recommended practice but helps to illustrate the idea of blocks and their ability to return values:

```
sqrt(4.0 +
        a:=3.0
        b:=1.0
        c:=a + b
        c
    )
```

$$2.8284271247\ 461900976$$

```
                                    Type:  Float
```

Note that indentation is **extremely** important. If the example above had the pile starting at "a:=" moved left by two spaces so that the "a" was under the "(" of the first line then the interpreter would signal an error. Furthermore if the closing parenthesis ")" is moved up to give

```
sqrt(4.0 +
        a:=3.0
        b:=1.0
        c:=a + b
        c)

  Line   1: sqrt(4.0 +
            ....A
  Error  A: Missing mate.
  Line   2:          a:=3.0
```

```
Line   3:            b:=1.0
Line   4:            c:=a + b
Line   5:            c)
           .........AB
Error  A: (from A up to B) Ignored.
Error  B: Improper syntax.
Error  B: syntax error at top level
Error  B: Possibly missing a )
  5 error(s) parsing
```

then the parser will generate errors. If the parenthesis is shifted right by several spaces so that it is in line with the "c" thus:

```
sqrt(4.0 +
        a:=3.0
        b:=1.0
        c:=a + b
        c
        )
```

```
Line   1: sqrt(4.0 +
           ....A
Error  A: Missing mate.
Line   2:            a:=3.0
Line   3:            b:=1.0
Line   4:            c:=a + b
Line   5:            c
Line   6:            )
           .........A
Error  A: (from A up to A) Ignored.
Error  A: Improper syntax.
Error  A: syntax error at top level
Error  A: Possibly missing a )
  5 error(s) parsing
```

a similar error will be raised. Finally, the ")" must be indented by at least one space relative to the sqrt thus:

```
sqrt(4.0 +
        a:=3.0
        b:=1.0
        c:=a + b
        c
 )
```

$$2.8284271247\ 461900976$$

```
                                          Type:  Float
```

or an error will be generated.

It can be seen that great care needs to be taken when constructing input files consisting of piles of expressions. It would seem prudent to add one pile at a time and check if it is acceptable before adding more, particularly if piles are nested. However, it should be pointed out that the use of piles as values for functions is not very readable and so perhaps the delicate nature of their interpretation should deter programmers from using them in these situations. Using piles should really be restricted to constructing functions, etc. and a small amount of rewriting can remove the need to use them as arguments. For example, the previous block could easily be implemented as:

```
a:=3.0
b:=1.0
c:=a + b
sqrt(4.0 + c)

a:=3.0
```

$$3.0$$

```
                                          Type:  Float
```

```
b:=1.0
```

$$1.0$$

```
                                          Type:  Float
```

```
c:=a + b
```

$$4.0$$

```
                                          Type:  Float
```

```
sqrt(4.0 + c)
```

$$2.8284271247\ 461900976$$

```
                                          Type:  Float
```

which achieves the same result and is easier to understand. Note that this is still a pile but it is not as fragile as the previous version.

0.6.3 Functions

Definitions of functions in Axiom are quite simple providing two things are observed. First, the type of the function must either be completely specified or completely unspecified. Second, the body of the function is assigned to the function identifier using the delayed assignment operator "==".

To specify the type of something the ":" operator is used. Thus to define a variable x to be of type `Fraction Integer` we enter:

```
x :  Fraction Integer
```

                                                                                   Type: Void

For functions the method is the same except that the arguments are placed in parentheses and the return type is placed after the symbol "->". Some examples of function definitions taking zero, one, two, or three arguments and returning a list of integers are:

```
f :  () -> List Integer
```

                                                                                   Type: Void

```
g :  (Integer) -> List Integer
```

                                                                                   Type: Void

```
h :  (Integer, Integer) -> List Integer
```

                                                                                   Type: Void

```
k :  (Integer, Integer, Integer) -> List Integer
```

                                                                                   Type: Void

Now the actual function definitions might be:

```
f() == [ ]
```

```
                                                     Type:   Void
```

```
g(a) == [a]
```

```
                                                     Type:   Void
```

```
h(a,b) == [a,b]
```

```
                                                     Type:   Void
```

```
k(a,b,c) == [a,b,c]
```

```
                                                     Type:   Void
```

with some invocations of these functions:

```
f()
```

```
   Compiling function f with type () -> List Integer
```

$$[\,]$$

```
                                         Type:   List Integer
```

```
g(4)
```

```
   Compiling function g with type Integer -> List Integer
```

$$[4]$$

```
                                         Type:   List Integer
```

```
h(2,9)
```

```
   Compiling function h with type (Integer,Integer) -> List Integer
```

$$[2, 9]$$

<div align="right">Type: List Integer</div>

```
k(-3,42,100)
```

```
   Compiling function k with type (Integer,Integer,Integer) -> List
      Integer
```

$$[-3, 42, 100]$$

<div align="right">Type: List Integer</div>

The value returned by a function is either the value of the last expression evaluated or the result of a **return** statement. For example, the following are effectively the same:

```
p :  Integer -> Integer
```

<div align="right">Type: Void</div>

```
p x == (a:=1; b:=2; a+b+x)
```

<div align="right">Type: Void</div>

```
p x == (a:=1; b:=2; return(a+b+x))
```

<div align="right">Type: Void</div>

Note that a block (pile) is assigned to the function identifier **p** and thus all the rules about blocks apply to function definitions. Also there was only one argument so the parenthese are not needed.

This is basically all that one needs to know about defining functions in Axiom – first specify the complete type and then assign a block to the function name. The rest of this section is concerned with defining more complex blocks than those in this section and as a result function definitions will crop up continually particularly since they are a good way of testing examples. Since the block structure is more complex we will use the **pile** notation and thus have to use input files to read the piles.

0.6.4 Choices

Apart from the "**=>**" operator that allows a block to exit before the end Axiom provides the standard **if-then-else** construct. The general syntax is:

if *BooleanExpr* then *Expr1* else *Expr2*

where "else *Expr2*" can be omitted. If the expression *BooleanExpr* evaluates to `true` then *Expr1* is executed otherwise *Expr2* (if present) will be executed. An example of piles and **if-then-else** is: (read from an input file)

```
h := 2.0
if h > 3.1 then
      1.0
   else
      z:= cos(h)
      max(x,0.5)


h := 2.0
```

$$2.0$$

```
                                              Type:  Float
```

```
if h > 3.1 then
      1.0
   else
      z:= cos(h)
      max(x,0.5)
```

$$x$$

```
                              Type:  Polynomial Float
```

Note the indentation – the "else" must be indented relative to the "if" otherwise it will generate an error (Axiom will think there are two piles, the second one beginning with "else").

Any expression that has type `Boolean` can be used as `BooleanExpr` and the most common will be those involving the relational operators ">", "<", and "=". Usually the type of an expression involving the equality operator "=" will be **Boolean** but in those situations when it isn't you may need to use the "@" operator to ensure that it is.

0.6.5   Loops

Loops in Axiom are regarded as expressions containing another expression called the *loop body*. The loop body is executed zero or more times depending on the kind of loop. Loops can be nested to any depth.

The repeat loop

The simplest kind of loop provided by Axiom is the **repeat** loop. The general syntax of this is:

repeat *loopBody*

This will cause Axiom to execute *loopBody* repeatedly until either a **break** or **return** statement is encountered. If *loopBody* contains neither of these statements then it will loop forever. The following piece of code will display the numbers from 1 to 4:

```
i:=1
repeat
  if i > 4 then break
  output(i)
  i:=i+1
```

```
i:=1
```

                                          1

                                                            Type: PositiveInteger

```
repeat
  if i > 4 then break
  output(i)
  i:=i+1
```

```
  1
  2
  3
  4
```

                                                                    Type: Void

It was mentioned that loops will only be left when either a **break** or **return** statement is encountered so why can't one use the "=>" operator? The reason

is that the "=>" operator tells Axiom to leave the current block whereas **break** leaves the current loop. The **return** statement leave the current function.

To skip the rest of a loop body and continue the next iteration of the loop use the **iterate** statement (the – starts a comment in Axiom)

```
i := 0
repeat
  i := i + 1
  if i > 6 then break
  -- Return to start if i is odd
  if odd?(i) then iterate
  output(i)

i := 0
```

$$0$$

                                    Type: NonNegativeInteger

```
repeat
  i := i + 1
  if i > 6 then break
  -- Return to start if i is odd
  if odd?(i) then iterate
  output(i)
```

```
  2
  4
  6
```

                                             Type: Void

The `while` loop

The while statement extends the basic **repeat** loop to place the control of leaving the loop at the start rather than have it buried in the middle. Since the body of the loop is still part of a **repeat** loop, **break** and "=>" work in the same way as in the previous section. The general syntax of a **while** loop is:

while *BoolExpr* repeat *loopBody*

As before, *BoolExpr* must be an expression of type **Boolean**. Before the body of the loop is executed *BoolExpr* is tested. If it evaluates to `true` then the loop body is entered otherwise the loop is terminated. Multiple conditions can be applied using the logical operators such as **and** or by using several **while** statements before the **repeat**.

```
x:=1
y:=1
while x < 4 and y < 10 repeat
  output [x,y]
  x := x + 1
  y := y + 2

x:=1
```

$$1$$

                                                                    Type: PositiveInteger

```
y:=1
```

$$1$$

                                                                    Type: PositiveInteger

```
while x < 4 and y < 10 repeat
  output [x,y]
  x := x + 1
  y := y + 2

    [1,1]
    [2,3]
    [3,5]
```

                                                                       Type: Void

```
x:=1
y:=1
while x < 4 while y < 10 repeat
  output [x,y]
  x := x + 1
  y := y + 2

x:=1
```

$$1$$

                                                                    Type: PositiveInteger

```
y:=1
```

$$1$$

<div align="right">Type: PositiveInteger</div>

```
while x < 4 while y < 10 repeat
  output [x,y]
  x := x + 1
  y := y + 2

   [1,1]
   [2,3]
   [3,5]
```

<div align="right">Type: Void</div>

Note that the last example using two **while** statements is *not* a nested loop but the following one is:

```
x:=1
y:=1
while x < 4 repeat
  while y < 10 repeat
    output [x,y]
    x := x + 1
    y := y + 2

x:=1
\begin{verbatim}
$$
1
$$
\returnType{Type: PositiveInteger}

\begin{verbatim}
y:=1
```

$$1$$

<div align="right">Type: PositiveInteger</div>

```
while x < 4 repeat
  while y < 10 repeat
    output [x,y]
    x := x + 1
    y := y + 2

  [1,1]
  [2,3]
  [3,5]
  [4,7]
  [5,9]
```

                                                                    Type: Void

Suppose we that, given a matrix of arbitrary size, find the position and value of
the first negative element by examining the matrix in row-major order:

```
m := matrix [ [ 21, 37, 53, 14 ],_
              [  8, 22,-24, 16 ],_
              [  2, 10, 15, 14 ],_
              [ 26, 33, 55,-13 ] ]
```

```
lastrow := nrows(m)
lastcol := ncols(m)
r := 1
while r <= lastrow repeat
  c := 1 -- Index of first column
  while c <= lastcol repeat
    if elt(m,r,c) < 0 then
      output [r,c,elt(m,r,c)]
      r := lastrow
      break -- Don't look any further
    c := c + 1
  r := r + 1
```

```
m := matrix [ [ 21, 37, 53, 14 ],_
              [  8, 22,-24, 16 ],_
              [  2, 10, 15, 14 ],_
              [ 26, 33, 55,-13 ] ]
```

$$
\begin{bmatrix}
21 & 37 & 53 & 14 \\
8 & 22 & -24 & 16 \\
2 & 10 & 15 & 14 \\
26 & 33 & 55 & -13
\end{bmatrix}
$$

                                                                    Type: Matrix Integer

```
lastrow := nrows(m)
```

$$4$$

<div align="right">Type: PositiveInteger</div>

```
lastcol := ncols(m)
```

$$4$$

<div align="right">Type: PositiveInteger</div>

```
r := 1
```

$$1$$

<div align="right">Type: PositiveInteger</div>

```
while r <= lastrow repeat
  c := 1 -- Index of first column
  while c <= lastcol repeat
    if elt(m,r,c) < 0 then
      output [r,c,elt(m,r,c)]
      r := lastrow
      break -- Don't look any further
    c := c + 1
  r := r + 1

  [2,3,- 24]
```

<div align="right">Type: Void</div>

The `for` loop

The last loop statement of interest is the **for** loop. There are two ways of creating a **for** loop. The first way uses either a list or a segment:

<div align="center">

for *var* in *seg* repeat *loopBody*
for *var* in *list* repeat *loopBody*

</div>

where *var* is an index variable which is iterated over the values in *seg* or *list*. The value *seg* is a segment such as $1 \ldots 10$ or $1 \ldots$ and *list* is a list of some type. For example:

```
for i in 1..10 repeat
  ~prime?(i) => iterate
  output(i)

for i in 1..10 repeat
  ~prime?(i) => iterate
  output(i)

   2
   3
   5
   7
```

<div align="right">Type: Void</div>

```
for w in ["This", "is", "your", "life!"] repeat
  output(w)

for w in ["This", "is", "your", "life!"] repeat
  output(w)

   This
   is
   your
   life!
```

<div align="right">Type: Void</div>

The second form of the **for** loop syntax includes a "**such that**" clause which must be of type **Boolean**:

$$for\ var — BoolExpr\ in\ seg\ repeat\ loopBody$$
$$for\ var — BoolExpr\ in\ list\ repeat\ loopBody$$

Some examples are:

```
for i in 1..10 | prime?(i) repeat
  output(i)

for i in 1..10 | prime?(i) repeat
  output(i)

   2
   3
   5
   7
```

```
                                                        Type:   Void
```

```
for i in [1,2,3,4,5,6,7,8,9,10] | prime?(i) repeat
  output(i)
```

```
for i in [1,2,3,4,5,6,7,8,9,10] | prime?(i) repeat
  output(i)

   2
   3
   5
   7
```

```
                                                        Type:   Void
```

You can also use a **while** clause:

```
for i in 1.. while i < 7 repeat
  if even?(i) then output(i)
```

```
for i in 1.. while i < 7 repeat
  if even?(i) then output(i)

   2
   4
   6
```

```
                                                        Type:   Void
```

Using the "**such that**" clause makes this appear simpler:

```
for i in 1.. | even?(i) while i < 7 repeat
  output(i)
\{verbatim}
\begin{verbatim}
for i in 1.. | even?(i) while i < 7 repeat
  output(i)

   2
   4
   6
```

```
                                                        Type:   Void
```

You can use multiple **for** clauses to iterate over several sequences in parallel:

```
for a in 1..4 for b in 5..8 repeat
  output [a,b]
```

```
for a in 1..4 for b in 5..8 repeat
  output [a,b]

   [1,5]
   [2,6]
   [3,7]
   [4,8]
```

```
                                             Type:  Void
```

As a general point it should be noted that any symbols referred to in the "**such that**" and **while** clauses must be pre-defined. This either means that the symbols must have been defined in an outer level (e.g. in an enclosing loop) or in a **for** clause appearing before the "**such that**" or **while**. For example:

```
for a in 1..4 repeat
  for b in 7..9 | prime?(a+b) repeat
    output [a,b,a+b]
```

```
for a in 1..4 repeat
  for b in 7..9 | prime?(a+b) repeat
    output [a,b,a+b]

   [2,9,11]
   [3,8,11]
   [4,7,11]
   [4,9,13]
```

```
                                             Type:  Void
```

Finally, the **for** statement has a **by** clause to specify the step size. This makes it possible to iterate over the segment in reverse order:

```
for a in 1..4 for b in 8..5 by -1 repeat
  output [a,b]
```

```
for a in 1..4 for b in 8..5 by -1 repeat
  output [a,b]
```

```
[1,8]
[2,7]
[3,6]
[4,5]
```

                                                    Type: Void

Note that without the "by -1" the segment 8..5 is empty so there is nothing to
iterate over and the loop exits immediately.

Chapter 1

An Overview of Axiom

> When we start cataloging the gains in tools sitting on a computer,
> the benefits of software are amazing. But, if the benefits of software
> are so great, why do we worry about making it easier – don't the
> ends pay for the means? We worry becuase making such software
> is extraordinarily hard and almost no one can do it – the detail is
> exhausting, the creativity required is extreme, the hours of failure
> upon failure requiring patience and persistence would tax anyone
> claiming to be sane. Yet we require people with such characteristics
> be found and employed and employed cheaply.
>
> – Christopher Alexander
>
> (from Patterns of Software by Richard Gabriel)

Welcome to the Axiom environment for interactive computation and problem
solving. Consider this chapter a brief, whirlwind tour of the Axiom world. We
introduce you to Axiom's graphics and the Axiom language. Then we give a
sampling of the large variety of facilities in the Axiom system, ranging from
the various kinds of numbers, to data types (like lists, arrays, and sets) and
mathematical objects (like matrices, integrals, and differential equations). We
conclude with the discussion of system commands and an interactive "undo."

Before embarking on the tour, we need to brief those readers working interac-
tively with Axiom on some details.

1.1   Starting Up and Winding Down

You need to know how to start the Axiom system and how to stop it. We
assume that Axiom has been correctly installed on your machine (as described
in another Axiom document).

To begin using Axiom, issue the command **axiom** to the Axiom operating system shell. There is a brief pause, some start-up messages, and then one or more windows appear.

If you are not running Axiom under the X Window System, there is only one window (the console). At the lower left of the screen there is a prompt that looks like

```
(1) ->
```

When you want to enter input to Axiom, you do so on the same line after the prompt. The "1" in "(1)", also called the equation number, is the computation step number and is incremented after you enter Axiom statements. Note, however, that a system command such as `)clear all` may change the step number in other ways. We talk about step numbers more when we discuss system commands and the workspace history facility.

If you are running Axiom under the X Window System, there may be two windows: the console window (as just described) and the HyperDoc main menu. HyperDoc is a multiple-window hypertext system that lets you view Axiom documentation and examples on-line, execute Axiom expressions, and generate graphics. If you are in a graphical windowing environment, it is usually started automatically when Axiom begins. If it is not running, issue `)hd` to start it. We discuss the basics of HyperDoc in Chapter 3 on page 175.

To interrupt an Axiom computation, hold down the **Ctrl** (control) key and press **c**. This brings you back to the Axiom prompt.

> To exit from Axiom, move to the console window, type `)quit` at the input prompt and press the **Enter** key. You will probably be prompted with the following message:
>
> Please enter **y** or **yes** if you really want to leave the
> interactive environment and return to the operating system
>
> You should respond **yes**, for example, to exit Axiom.

We are purposely vague in describing exactly what your screen looks like or what messages Axiom displays. Axiom runs on a number of different machines, operating systems and window environments, and these differences all affect the physical look of the system. You can also change the way that Axiom behaves via *system commands* described later in this chapter and in Appendix A. System commands are special commands, like `)set`, that begin with a closing parenthesis and are used to change your environment. For example, you can set a system variable so that you are not prompted for confirmation when you want to leave Axiom.

1.1.1 Clef

If you are using Axiom under the X Window System, the Clef command line editor is probably available and installed. With this editor you can recall previous lines with the up and down arrow keys. To move forward and backward on a line, use the right and left arrows. You can use the **Insert** key to toggle insert mode on or off. When you are in insert mode, the cursor appears as a large block and if you type anything, the characters are inserted into the line without deleting the previous ones.

If you press the **Home** key, the cursor moves to the beginning of the line and if you press the **End** key, the cursor moves to the end of the line. Pressing **Ctrl-End** deletes all the text from the cursor to the end of the line.

Clef also provides Axiom operation name completion for a limited set of operations. If you enter a few letters and then press the **Tab** key, Clef tries to use those letters as the prefix of an Axiom operation name. If a name appears and it is not what you want, press **Tab** again to see another name.

You are ready to begin your journey into the world of Axiom.

1.2 Typographic Conventions

In this document we have followed these typographical conventions:

- Categories, domains and packages are displayed in this font: `Ring`, `Integer`, `DiophantineSolutionPackage`.

- Prefix operators, infix operators, and punctuation symbols in the Axiom language are displayed in the text like this: `+`, `$`, `+->`.

- Axiom expressions or expression fragments are displayed in this font: `inc(x) == x + 1`.

- For clarity of presentation, TeX is often used to format expressions $g(x) = x^2 + 1$.

- Function names and HyperDoc button names are displayed in the text in this font: **factor**, **integrate**, **Lighting**.

- Italics are used for emphasis and for words defined in the glossary: *category*.

This document contains over 2500 examples of Axiom input and output. All examples were run though Axiom and their output was created in TeX form by the Axiom `TexFormat` package. We have deleted system messages from the example output if those messages are not important for the discussions in which the examples appear.

1.3 The Axiom Language

The Axiom language is a rich language for performing interactive computations and for building components of the Axiom library. Here we present only some basic aspects of the language that you need to know for the rest of this chapter. Our discussion here is intentionally informal, with details unveiled on an "as needed" basis. For more information on a particular construct, we suggest you consult the index.

1.3.1 Arithmetic Expressions

For arithmetic expressions, use the "+" and "-" operator as in mathematics. Use "*" for multiplication, and "**" for exponentiation. To create a fraction, use "/". When an expression contains several operators, those of highest *precedence* are evaluated first. For arithmetic operators, "**" has highest precedence, "*" and "/" have the next highest precedence, and "+" and "-" have the lowest precedence.

Axiom puts implicit parentheses around operations of higher precedence, and groups those of equal precedence from left to right.

```
1 + 2 - 3 / 4 * 3 ** 2 - 1
```

$$-\frac{19}{4}$$

<div align="right">Type: Fraction Integer</div>

The above expression is equivalent to this.

```
((1 + 2) - ((3 / 4) * (3 ** 2))) - 1
```

$$-\frac{19}{4}$$

<div align="right">Type: Fraction Integer</div>

If an expression contains subexpressions enclosed in parentheses, the parenthesized subexpressions are evaluated first (from left to right, from inside out).

```
1 + 2 - 3/ (4 * 3 ** (2 - 1))
```

$$\frac{11}{4}$$

<div align="right">Type: Fraction Integer</div>

1.3.2   Previous Results

Use the percent sign "**%**" to refer to the last result. Also, use "**%%**' to refer
to previous results. "**%%(-1)**" is equivalent to "**%**", "**%%(-2)**" returns the next
to the last result, and so on. "**%%(1)**" returns the result from step number
1, "**%%(2)**" returns the result from step number 2, and so on. "**%%(0)**" is not
defined.

This is ten to the tenth power.

```
10 ** 10
```

<div align="center">

10000000000

</div>

<div align="right">

Type: PositiveInteger

</div>

This is the last result minus one.

```
% - 1
```

<div align="center">

9999999999

</div>

<div align="right">

Type: PositiveInteger

</div>

This is the last result.

```
%%(-1)
```

<div align="center">

9999999999

</div>

<div align="right">

Type: PositiveInteger

</div>

This is the result from step number 1.

```
%%(1)
```

<div align="center">

10000000000

</div>

<div align="right">

Type: PositiveInteger

</div>

1.3.3 Some Types

Everything in Axiom has a type. The type determines what operations you can perform on an object and how the object can be used. Chapter 2 on page 129 is dedicated to the interactive use of types. Several of the final chapters discuss how types are built and how they are organized in the Axiom library.

Positive integers are given type **PositiveInteger**.

8

$$8$$

                                  Type: PositiveInteger

Negative ones are given type **Integer**. This fine distinction is helpful to the Axiom interpreter.

-8

$$-8$$

                                        Type: Integer

Here a positive integer exponent gives a polynomial result.

x**8

$$x^8$$

                                  Type: Polynomial Integer

Here a negative integer exponent produces a fraction.

x**(-8)

$$\frac{1}{x^8}$$

                              Type: Fraction Polynomial Integer

1.3.4 Symbols, Variables, Assignments, and Declarations

A *symbol* is a literal used for the input of things like the "variables" in polynomials and power series.

We use the three symbols x, y, and z in entering this polynomial.

```
(x - y*z)**2
```

$$y^2 \, z^2 - 2 \, x \, y \, z + x^2$$

<div align="right">Type: Polynomial Integer</div>

A symbol has a name beginning with an uppercase or lowercase alphabetic character, "%", or "!". Successive characters (if any) can be any of the above, digits, or "?". Case is distinguished: the symbol `points` is different from the symbol `Points`.

A symbol can also be used in Axiom as a *variable*. A variable refers to a value. To *assign* a value to a variable, the operator ":=" is used.[1] A variable initially has no restrictions on the kinds of values to which it can refer.

This assignment gives the value 4 (an integer) to a variable named x.

```
x := 4
```

<div align="center">4</div>

<div align="right">Type: PositiveInteger</div>

This gives the value $z + 3/5$ (a polynomial) to x.

```
x := z + 3/5
```

$$z + \frac{3}{5}$$

<div align="right">Type: Polynomial Fraction Integer</div>

To restrict the types of objects that can be assigned to a variable, use a *declaration*

```
y :  Integer
```

---

[1]Axiom actually has two forms of assignment: *immediate* assignment, as discussed here, and *delayed assignment*. See Section 5.1 on page 195 for details.

```
                                              Type:   Void
```

After a variable is declared to be of some type, only values of that type can be
assigned to that variable.

```
y := 89
```

$$89$$

```
                                              Type:   Integer
```

The declaration for y forces values assigned to y to be converted to integer
values.

```
y := sin %pi
```

$$0$$

```
                                              Type:   Integer
```

If no such conversion is possible, Axiom refuses to assign a value to y.

```
y := 2/3
```

```
   Cannot convert right-hand side of assignment
   2
   -
   3

      to an object of the type Integer of the left-hand side.
```

A type declaration can also be given together with an assignment. The decla-
ration can assist Axiom in choosing the correct operations to apply.

```
f :  Float := 2/3
```

$$0.6666666666\ 6666666667$$

```
                                              Type:   Float
```

Any number of expressions can be given on input line. Just separate them by
semicolons. Only the result of evaluating the last expression is displayed.

These two expressions have the same effect as the previous single expression.

```
f :  Float; f := 2/3
```

$$0.6666666666\ 6666666667$$

<div align="right">Type: Float</div>

The type of a symbol is either **Symbol** or **Variable**(*name*) where *name* is the name of the symbol.

By default, the interpreter gives this symbol the type **Variable(q)**.

```
q
```

$$q$$

<div align="right">Type: Variable q</div>

When multiple symbols are involved, **Symbol** is used.

```
[q, r]
```

$$[q, r]$$

<div align="right">Type: List OrderedVariableList [q,r]</div>

What happens when you try to use a symbol that is the name of a variable?

```
f
```

$$0.6666666666\ 6666666667$$

<div align="right">Type: Float</div>

Use a single quote "'" before the name to get the symbol.

```
'f
```

$$f$$

<div align="right">Type: Variable f</div>

Quoting a name creates a symbol by preventing evaluation of the name as a variable. Experience will teach you when you are most likely going to need to use a quote. We try to point out the location of such trouble spots.

1.3.5   Conversion

Objects of one type can usually be "converted" to objects of several other types. To *convert* an object to a new type, use the "::" infix operator.[2] For example, to display an object, it is necessary to convert the object to type `OutputForm`.

This produces a polynomial with rational number coefficients.

```
p := r**2 + 2/3
```

$$r^2 + \frac{2}{3}$$

Type: Polynomial Fraction Integer

Create a quotient of polynomials with integer coefficients by using "::".

```
p ::  Fraction Polynomial Integer
```

$$\frac{3\ r^2 + 2}{3}$$

Type: Fraction Polynomial Integer

Some conversions can be performed automatically when Axiom tries to evaluate your input. Others conversions must be explicitly requested.

1.3.6   Calling Functions

As we saw earlier, when you want to add or subtract two values, you place the arithmetic operator "+" or "-" between the two arguments denoting the values. To use most other Axiom operations, however, you use another syntax: write the name of the operation first, then an open parenthesis, then each of the arguments separated by commas, and, finally, a closing parenthesis. If the operation takes only one argument and the argument is a number or a symbol, you can omit the parentheses.

This calls the operation **factor** with the single integer argument 120.

```
factor(120)
```

$$2^3\ 3\ 5$$

Type: Factored Integer

---

[2]Conversion is discussed in detail in 2.7 on page 155.

This is a call to **divide** with the two integer arguments 125 and 7.

```
divide(125,7)
```

$$[quotient = 17, remainder = 6]$$

```
                    Type:  Record(quotient:  Integer, remainder:  Integer)
```

This calls **quatern** with four floating-point arguments.

```
quatern(3.4,5.6,2.9,0.1)
```

$$3.4 + 5.6 \ i + 2.9 \ j + 0.1 \ k$$

```
                                        Type:   Quaternion Float
```

This is the same as **factorial**(10).

```
factorial 10
```

$$3628800$$

```
                                        Type:  PositiveInteger
```

An operations that returns a `Boolean` value (that is, `true` or `false`) frequently has a name suffixed with a question mark ("?"). For example, the **even?** operation returns `true` if its integer argument is an even number, `false` otherwise.

An operation that can be destructive on one or more arguments usually has a name ending in a exclamation point ("!"). This actually means that it is *allowed* to update its arguments but it is not *required* to do so. For example, the underlying representation of a collection type may not allow the very last element to removed and so an empty object may be returned instead. Therefore, it is important that you use the object returned by the operation and not rely on a physical change having occurred within the object. Usually, destructive operations are provided for efficiency reasons.

1.3.7   Some Predefined Macros

Axiom provides several macros for your convenience.[3] Macros are names (or forms) that expand to larger expressions for commonly used values.

---

[3]See 6.2 on page 232 for a discussion on how to write your own macros.

%i	The square root of -1.
%e	The base of the natural logarithm.
%pi	π.
%infinity	∞.
%plusInfinity	$+\infty$.
%minusInfinity	$-\infty$.

To display all the macros (along with anything you have defined in the workspace), issue the system command `)display all`.

1.3.8   Long Lines

When you enter Axiom expressions from your keyboard, there will be times when they are too long to fit on one line. Axiom does not care how long your lines are, so you can let them continue from the right margin to the left side of the next line.

Alternatively, you may want to enter several shorter lines and have Axiom glue them together. To get this glue, put an underscore (`_`) at the end of each line you wish to continue.

```
2_
+_
3
```

is the same as if you had entered

```
2+3
```

Axiom statements in an input file (see Section 4.1 on page 183), can use indentation to indicate the program structure . (see Section 5.2 on page 199).

1.3.9   Comments

Comment statements begin with two consecutive hyphens or two consecutive plus signs and continue until the end of the line.

The comment beginning with "`--`" is ignored by Axiom.

```
2 + 3 -- this is rather simple, no?
```

$$5$$

```
                                    Type:  PositiveInteger
```

There is no way to write long multi-line comments other than starting each line with "`--`" or "`++`".

1.4 Numbers

Axiom distinguishes very carefully between different kinds of numbers, how they are represented and what their properties are. Here are a sampling of some of these kinds of numbers and some things you can do with them.

Integer arithmetic is always exact.

```
11**13 * 13**11 * 17**7 - 19**5 * 23**3
```

$$25387751125389185946662244484237298$$

<div align="right">Type: PositiveInteger</div>

Integers can be represented in factored form.

```
factor 6432380707485690237205944125517043441455570763243
```

$$11^{13} \ 13^{11} \ 17^7 \ 19^5 \ 23^3 \ 29^2$$

<div align="right">Type: Factored Integer</div>

Results stay factored when you do arithmetic. Note that the 12 is automatically factored for you.

```
% * 12
```

$$2^2 \ 3 \ 11^{13} \ 13^{11} \ 17^7 \ 19^5 \ 23^3 \ 29^2$$

<div align="right">Type: Factored Integer</div>

Integers can also be displayed to bases other than 10. This is an integer in base 11.

```
radix(25937424601,11)
```

$$10000000000$$

<div align="right">Type: RadixExpansion 11</div>

Roman numerals are also available for those special occasions.

```
roman(1992)
```

$$MCMXCII$$

<div align="right">Type: RomanNumeral</div>

Rational number arithmetic is also exact.

```
r := 10 + 9/2 + 8/3 + 7/4 + 6/5 + 5/6 + 4/7 + 3/8 + 2/9
```

$$\frac{55739}{2520}$$

<div align="right">Type: Fraction Integer</div>

To factor fractions, you have to pmap **factor** onto the numerator and denominator.

```
map(factor,r)
```

$$\frac{139\ 401}{2^3\ 3^2\ 5\ 7}$$

<div align="right">Type: Fraction Factored Integer</div>

`SingleInteger` refers to machine word-length integers.

In English, this expression means "11 as a small integer".

```
11@SingleInteger
```

<div align="center">11</div>

<div align="right">Type: SingleInteger</div>

Machine double-precision floating-point numbers are also available for numeric and graphical applications.

```
123.21@DoubleFloat
```

<div align="center">123.21000000000001</div>

<div align="right">Type: DoubleFloat</div>

The normal floating-point type in Axiom, `Float`, is a software implementation of floating-point numbers in which the exponent and the mantissa may have any number of digits. The types `Complex(Float)` and `Complex(DoubleFloat)` are the corresponding software implementations of complex floating-point numbers.

This is a floating-point approximation to about twenty digits. The "::" is used here to change from one kind of object (here, a rational number) to another (a floating-point number).

```
r ::  Float
```

$$22.118650793650793651$$

<div align="right">Type: Float</div>

Use **digits** to change the number of digits in the representation. This operation returns the previous value so you can reset it later.

```
digits(22)
```

$$20$$

<div align="right">Type: PositiveInteger</div>

To 22 digits of precision, the number $e^{\pi\sqrt{163.0}}$ appears to be an integer.

```
exp(%pi * sqrt 163.0)
```

$$262537412640768744.0$$

<div align="right">Type: Float</div>

Increase the precision to forty digits and try again.

```
digits(40); exp(%pi * sqrt 163.0)
```

$$26253741\ 2640768743.9999999999\ 9925007259\ 76$$

<div align="right">Type: Float</div>

Here are complex numbers with rational numbers as real and imaginary parts.

```
(2/3 + %i)**3
```

$$-\frac{46}{27} + \frac{1}{3}\,i$$

<div align="right">Type: Complex Fraction Integer</div>

The standard operations on complex numbers are available.

```
conjugate %
```

$$-\frac{46}{27} - \frac{1}{3} \ i$$

<div align="right">Type: Complex Fraction Integer</div>

You can factor complex integers.

```
factor(89 - 23 * %i)
```

$$-(1 + i) \ (2 + i)^2 \ (3 + 2 \ i)^2$$

<div align="right">Type: Factored Complex Integer</div>

Complex numbers with floating point parts are also available.

```
exp(%pi/4.0 * %i)
```

0.7071067811 8654752440 0844362104 8490392849+

0.7071067811 8654752440 0844362104 8490392848 i

<div align="right">Type: Complex Float</div>

The real and imaginary parts can be symbolic.

```
complex(u,v)
```

$$u + v \ i$$

<div align="right">Type: Complex Polynomial Integer</div>

Of course, you can do complex arithmetic with these also.

```
% ** 2
```

$$-v^2 + u^2 + 2 \ u \ v \ i$$

<div align="right">Type: Complex Polynomial Integer</div>

Every rational number has an exact representation as a repeating decimal expansion

```
decimal(1/352)
```

$$0.0028\overline{409}$$

A rational number can also be expressed as a continued fraction.

```
continuedFraction(6543/210)
```

$$31 + \frac{1|}{|6} + \frac{1|}{|2} + \frac{1|}{|1} + \frac{1|}{|3}$$

Type: ContinuedFraction Integer

Also, partial fractions can be used and can be displayed in a compact format

```
partialFraction(1,factorial(10))
```

$$\frac{159}{2^8} - \frac{23}{3^4} - \frac{12}{5^2} + \frac{1}{7}$$

Type: PartialFraction Integer

or expanded format.

```
padicFraction(%)
```

$$\frac{1}{2} + \frac{1}{2^4} + \frac{1}{2^5} + \frac{1}{2^6} + \frac{1}{2^7} + \frac{1}{2^8} - \frac{2}{3^2} - \frac{1}{3^3} - \frac{2}{3^4} - \frac{2}{5} - \frac{2}{5^2} + \frac{1}{7}$$

Type: PartialFraction Integer

Like integers, bases (radices) other than ten can be used for rational numbers. Here we use base eight.

```
radix(4/7, 8)
```

$$0.\overline{4}$$

Type: RadixExpansion 8

Of course, there are complex versions of these as well. Axiom decides to make the result a complex rational number.

```
% + 2/3*%i
```

$$\frac{4}{7} + \frac{2}{3} i$$

<div align="right">Type: Complex Fraction Integer</div>

You can also use Axiom to manipulate fractional powers.

`(5 + sqrt 63 + sqrt 847)**(1/3)`

$$\sqrt[3]{14\ \sqrt{7}+5}$$

<div align="right">Type: AlgebraicNumber</div>

You can also compute with integers modulo a prime.

`x :  PrimeField 7 := 5`

$$5$$

<div align="right">Type: PrimeField 7</div>

Arithmetic is then done modulo 7.

`x**3`

$$6$$

<div align="right">Type: PrimeField 7</div>

Since 7 is prime, you can invert nonzero values.

`1/x`

$$3$$

<div align="right">Type: PrimeField 7</div>

You can also compute modulo an integer that is not a prime.

`y :  IntegerMod 6 := 5`

$$5$$

<div align="right">Type: IntegerMod 6</div>

All of the usual arithmetic operations are available.

```
y**3
```

$$5$$

<div align="right">Type: IntegerMod 6</div>

Inversion is not available if the modulus is not a prime number. Modular arithmetic and prime fields are discussed in Section 8.11.1 on page 413.

```
1/y
```

```
   There are 12 exposed and 13 unexposed library operations named /
      having 2 argument(s) but none was determined to be applicable.
      Use HyperDoc Browse, or issue
                              )display op /
      to learn more about the available operations. Perhaps
      package-calling the operation or using coercions on the arguments
      will allow you to apply the operation.

   Cannot find a definition or applicable library operation named /
      with argument type(s)
                              PositiveInteger
                               IntegerMod 6

      Perhaps you should use "@" to indicate the required return type,
      or "$" to specify which version of the function you need.
```

This defines a to be an algebraic number, that is, a root of a polynomial equation.

```
a := rootOf(a**5 + a**3 + a**2 + 3,a)
```

$$a$$

<div align="right">Type: Expression Integer</div>

Computations with a are reduced according to the polynomial equation.

```
(a + 1)**10
```

$$-85\ a^4 - 264\ a^3 - 378\ a^2 - 458\ a - 287$$

<div align="right">Type: Expression Integer</div>

Define b to be an algebraic number involving a.

```
b := rootOf(b**4 + a,b)
```

$$b$$

<div align="right">Type: Expression Integer</div>

Do some arithmetic.

```
2/(b - 1)
```

$$\frac{2}{b-1}$$

<div align="right">Type: Expression Integer</div>

To expand and simplify this, call *ratDenom* to rationalize the denominator.

```
ratDenom(%)
```

$$\left(a^4 - a^3 + 2\,a^2 - a + 1\right)\,b^3 + \left(a^4 - a^3 + 2\,a^2 - a + 1\right)\,b^2 +$$
$$\left(a^4 - a^3 + 2\,a^2 - a + 1\right)\,b + a^4 - a^3 + 2\,a^2 - a + 1$$

<div align="right">Type: Expression Integer</div>

If we do this, we should get b.

```
2/%+1
```

$$\frac{\left(\begin{array}{l}\left(a^4 - a^3 + 2\,a^2 - a + 1\right)\,b^3 + \left(a^4 - a^3 + 2\,a^2 - a + 1\right)\,b^2 + \\ \left(a^4 - a^3 + 2\,a^2 - a + 1\right)\,b + a^4 - a^3 + 2\,a^2 - a + 3\end{array}\right)}{\left(\begin{array}{l}\left(a^4 - a^3 + 2\,a^2 - a + 1\right)\,b^3 + \left(a^4 - a^3 + 2\,a^2 - a + 1\right)\,b^2 + \\ \left(a^4 - a^3 + 2\,a^2 - a + 1\right)\,b + a^4 - a^3 + 2\,a^2 - a + 1\end{array}\right)}$$

<div align="right">Type: Expression Integer</div>

But we need to rationalize the denominator again.

```
ratDenom(%)
```

$$b$$

<div align="right">Type: Expression Integer</div>

Types `Quaternion` and `Octonion` are also available. Multiplication of quaternions is non-commutative, as expected.

```
q:=quatern(1,2,3,4)*quatern(5,6,7,8) -
quatern(5,6,7,8)*quatern(1,2,3,4)
```

$$-8\ i + 16\ j - 8\ k$$

<div align="right">Type: Quaternion Integer</div>

1.5 Data Structures

Axiom has a large variety of data structures available. Many data structures are particularly useful for interactive computation and others are useful for building applications. The data structures of Axiom are organized into *category hierarchies*.

A *list* [4] is the most commonly used data structure in Axiom for holding objects all of the same type. The name *list* is short for "linked-list of nodes." Each node consists of a value (**first**) and a link (**rest**) that points to the next node, or to a distinguished value denoting the empty list. To get to, say, the third element, Axiom starts at the front of the list, then traverses across two links to the third node.

Write a list of elements using square brackets with commas separating the elements.

```
u := [1,-7,11]
```

$$[1, -7, 11]$$

<div align="right">Type: List Integer</div>

This is the value at the third node. Alternatively, you can say $u.3$.

```
first rest rest u
```

<div align="center">11</div>

---

[4]Lists are discussed in Section 9.47 on page 675

                                                Type: PositiveInteger

Many operations are defined on lists, such as: **empty?**, to test that a list has
no elements; **cons**(x, l), to create a new list with **first** element x and **rest** l;
reverse, to create a new list with elements in reverse order; and **sort**, to arrange
elements in order.

An important point about lists is that they are "mutable": their constituent
elements and links can be changed "in place." To do this, use any of the
operations whose names end with the character "!".

The operation **concat!**(u, v) replaces the last link of the list u to point to some
other list v. Since u refers to the original list, this change is seen by u.

```
concat!(u,[9,1,3,-4]); u
```

$$[1, -7, 11, 9, 1, 3, -4]$$

                                                Type: List Integer

A *cyclic list* is a list with a "cycle": a link pointing back to an earlier node of
the list. To create a cycle, first get a node somewhere down the list.

```
lastnode := rest(u,3)
```

$$[9, 1, 3, -4]$$

                                                Type: List Integer

Use **setrest!** to change the link emanating from that node to point back to an
earlier part of the list.

```
setrest!(lastnode,rest(u,2)); u
```

$$\left[1, -7, \overline{11, 9}\right]$$

                                                Type: List Integer

A *stream* is a structure that (potentially) has an infinite number of distinct
elements. Think of a stream as an "infinite list" where elements are computed
successively. [5]

Create an infinite stream of factored integers. Only a certain number of initial
elements are computed and displayed.

_____

[5]Streams are discussed in SectionStreamXmpPage on page 801

```
[factor(i) for i in 2..  by 2]
```

$$\left[2, 2^2, 2\ 3, 2^3, 2\ 5, 2^2\ 3, 2\ 7, 2^4, 2\ 3^2, 2^2\ 5, \ldots\right]$$

<div align="right">Type: Stream Factored Integer</div>

Axiom represents streams by a collection of already-computed elements together with a function to compute the next element "on demand." Asking for the n-th element causes elements 1 through n to be evaluated.

```
%.36
```

$$2^3\ 3^2$$

<div align="right">Type: Factored Integer</div>

Streams can also be finite or cyclic. They are implemented by a linked list structure similar to lists and have many of the same operations. For example, **first** and **rest** are used to access elements and successive nodes of a stream.

A *one-dimensional array* is another data structure used to hold objects of the same type [6]. Unlike lists, one-dimensional arrays are inflexible—they are implemented using a fixed block of storage. Their advantage is that they give quick and equal access time to any element.

A simple way to create a one-dimensional array is to apply the operation **oneDimensionalArray** to a list of elements.

```
a := oneDimensionalArray [1, -7, 3, 3/2]
```

$$\left[1, -7, 3, \frac{3}{2}\right]$$

<div align="right">Type: OneDimensionalArray Fraction Integer</div>

One-dimensional arrays are also mutable: you can change their constituent elements "in place."

```
a.3 := 11; a
```

$$\left[1, -7, 11, \frac{3}{2}\right]$$

<div align="right">Type: OneDimensionalArray Fraction Integer</div>

---

[6]OnedimensionalArray is discussed in Section 9.57 on page 715

However, one-dimensional arrays are not flexible structures. You cannot destructively **concat!** them together.

```
concat!(a,oneDimensionalArray [1,-2])
```

```
   There are 5 exposed and 0 unexposed library operations named concat!
      having 2 argument(s) but none was determined to be applicable.
      Use HyperDoc Browse, or issue
                              )display op concat!
      to learn more about the available operations. Perhaps
      package-calling the operation or using coercions on the arguments
      will allow you to apply the operation.

   Cannot find a definition or applicable library operation named
      concat! with argument type(s)
                    OneDimensionalArray Fraction Integer
                         OneDimensionalArray Integer

      Perhaps you should use "@" to indicate the required return type,
      or "$" to specify which version of the function you need.
```

Examples of datatypes similar to `OneDimensionalArray` are: `Vector` (vectors are mathematical structures implemented by one-dimensional arrays), `String` (arrays of "characters," represented by byte vectors), and `Bits` (represented by "bit vectors").

A vector of 32 bits, each representing the **Boolean** value `true`.

```
bits(32,true)
```

$$"11111111111111111111111111111111"$$

```
                                                            Type:  Bits
```

A *flexible array* [7] is a cross between a list and a one-dimensional array. Like a one-dimensional array, a flexible array occupies a fixed block of storage. Its block of storage, however, has room to expand. When it gets full, it grows (a new, larger block of storage is allocated); when it has too much room, it contracts.

Create a flexible array of three elements.

```
f := flexibleArray [2, 7, -5]
```

$$[2, 7, -5]$$

---

[7]FlexibleArray is discussed in Section 9.26 on page 561

$$\text{Type: FlexibleArray Integer}$$

Insert some elements between the second and third elements.

```
insert!(flexibleArray [11, -3],f,2)
```

$$[2, 11, -3, 7, -5]$$

$$\text{Type: FlexibleArray Integer}$$

Flexible arrays are used to implement "heaps." A *heap* is an example of a data structure called a *priority queue*, where elements are ordered with respect to one another. A heap [8] is organized so as to optimize insertion and extraction of maximum elements. The **extract!** operation returns the maximum element of the heap, after destructively removing that element and reorganizing the heap so that the next maximum element is ready to be delivered.

An easy way to create a heap is to apply the operation *heap* to a list of values.

```
h := heap [-4,7,11,3,4,-7]
```

$$[11, 4, 7, -4, 3, -7]$$

$$\text{Type: Heap Integer}$$

This loop extracts elements one-at-a-time from h until the heap is exhausted, returning the elements as a list in the order they were extracted.

```
[extract!(h) while not empty?(h)]
```

$$[11, 7, 4, 3, -4, -7]$$

$$\text{Type: List Integer}$$

A *binary tree* is a "tree" with at most two branches per node: it is either empty, or else is a node consisting of a value, and a left and right subtree (again, binary trees). [9] Examples of binary tree types are `BinarySearchTree`, `PendantTree`, `TournamentTree`, and `BalancedBinaryTree`.

A *binary search tree* is a binary tree such that, for each node, the value of the node is greater than all values (if any) in the left subtree, and less than or equal all values (if any) in the right subtree.

```
binarySearchTree [5,3,2,9,4,7,11]
```

---

[8] Heap is discussed in Section 9.32 on page 585

[9] BinarySearchTrees are discussed in Section 9.5 on page 470

$$[[2, 3, 4], 5, [7, 9, 11]]$$

```
                        Type:  BinarySearchTree PositiveInteger
```

A *balanced binary tree* is useful for doing modular computations. Given a list
lm of moduli, **modTree**(a, lm) produces a balanced binary tree with the values
$a \bmod m$ at its leaves.

```
modTree(8,[2,3,5,7])
```

$$[0, 2, 3, 1]$$

```
                                        Type:  List Integer
```

A *set* is a collection of elements where duplication and order is irrelevant. [10] Sets
are always finite and have no corresponding structure like streams for infinite
collections.

Create sets using braces "{ " and "}" rather than brackets.

```
fs := set[1/3,4/5,-1/3,4/5]
```

$$\left\{ -\frac{1}{3}, \frac{1}{3}, \frac{4}{5} \right\}$$

```
                                    Type:  Set Fraction Integer
```

A *multiset* is a set that keeps track of the number of duplicate values. [11]

For all the primes p between 2 and 1000, find the distribution of $p \bmod 5$.

```
multiset [x rem 5 for x in primes(2,1000)]
```

$$\{0, 42\colon\ 3, 40\colon\ 1, 38\colon\ 4, 47\colon\ 2\}$$

```
                                        Type:  Multiset Integer
```

A *table* is conceptually a set of "key–value" pairs and is a generalization of a
multiset. For examples of tables, see `AssociationList`, `HashTable`, `KeyedAccessFile`,
`Library`, `SparseTable`, `StringTable`, and `Table`. The domain `Table(Key,`
`Entry)` provides a general-purpose type for tables with *values* of type *Entry*
indexed by *keys* of type *Key*.

Compute the above distribution of primes using tables. First, let t denote an
empty table of keys and values, each of type `Integer`.

---

[10]Sets are discussed in Section 9.71 on page 786
[11]Multisets are discussed in Section 9.53 on page 706

```
t :  Table(Integer,Integer) := empty()
```

$$\text{table}()$$

```
                                   Type:  Table(Integer,Integer)
```

We define a function **howMany** to return the number of values of a given modulus k seen so far. It calls **search**(k, t) which returns the number of values stored under the key k in table t, or ``failed'' if no such value is yet stored in t under k.

In English, this says "Define $howMany(k)$ as follows. First, let n be the value of $search(k, t)$. Then, if n has the value "$failed$", return the value 1; otherwise return $n + 1$."

```
howMany(k) == (n:=search(k,t); n case "failed" => 1; n+1)
```

```
                                                     Type:  Void
```

Run through the primes to create the table, then print the table. The expression `t.m := howMany(m)` updates the value in table t stored under key m.

```
for p in primes(2,1000) repeat (m:= p rem 5; t.m:= howMany(m)); t
```

```
   Compiling function howMany with type Integer -> Integer
```

$$\text{table} \left(2 = 47, 4 = 38, 1 = 40, 3 = 42, 0 = 1\right)$$

```
                                   Type:  Table(Integer,Integer)
```

A *record* is an example of an inhomogeneous collection of objects.[12] A record consists of a set of named *selectors* that can be used to access its components.

Declare that *daniel* can only be assigned a record with two prescribed fields.

```
daniel :  Record(age :  Integer, salary :  Float)
```

```
                                                     Type:  Void
```

Give *daniel* a value, using square brackets to enclose the values of the fields.

```
daniel := [28, 32005.12]
```

---

[12]See 2.4 on page 145 for details.

$$[age = 28, salary = 32005.12]$$

```
                         Type:  Record(age:  Integer,salary:  Float)
```

Give *daniel* a raise.

```
daniel.salary := 35000; daniel
```

$$[age = 28, salary = 35000.0]$$

```
                         Type:  Record(age:  Integer,salary:  Float)
```

A *union* is a data structure used when objects have multiple types.[13]

Let *dog* be either an integer or a string value.

```
dog:  Union(licenseNumber:  Integer, name:  String)
```

```
                                                       Type:  Void
```

Give *dog* a name.

```
dog := "Whisper"
```

```
                            "Whisper"
```

```
                                    Type:  Union(name:  String,...)
```

All told, there are over forty different data structures in Axiom. Using the
domain constructors described in Chapter 13 on page 923, you can add your
own data structure or extend an existing one. Choosing the right data structure
for your application may be the key to obtaining good performance.

1.6   Expanding to Higher Dimensions

To get higher dimensional aggregates, you can create one-dimensional aggregates
with elements that are themselves aggregates, for example, lists of lists, one-
dimensional arrays of lists of multisets, and so on. For applications requiring
two-dimensional homogeneous aggregates, you will likely find *two-dimensional
arrays* and *matrices* most useful.

The entries in `TwoDimensionalArray` and `Matrix` objects are all the same type,
except that those for `Matrix` must belong to a `Ring`. You create and access

---

[13]See 2.5 on page 149 for details.

elements in roughly the same way. Since matrices have an understood algebraic structure, certain algebraic operations are available for matrices but not for arrays. Because of this, we limit our discussion here to `Matrix`, that can be regarded as an extension of `TwoDimensionalArray`. See `TwoDimensionalArray` for more information about arrays. For more information about Axiom's linear algebra facilities, see `Matrix`, `Permanent`, `SquareMatrix`, `Vector`, see Section 8.4 on page 363 (computation of eigenvalues and eigenvectors), and Section 8.5 on page 366 (solution of linear and polynomial equations).

You can create a matrix from a list of lists, where each of the inner lists represents a row of the matrix.

```
m := matrix([ [1,2], [3,4] ])
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

<div align="right">Type: Matrix Integer</div>

The "collections" construct (see 5.5 on page 224) is useful for creating matrices whose entries are given by formulas.

```
matrix([ [1/(i + j - x) for i in 1..4] for j in 1..4])
```

$$\begin{bmatrix} -\frac{1}{x-2} & -\frac{1}{x-3} & -\frac{1}{x-4} & -\frac{1}{x-5} \\ -\frac{1}{x-3} & -\frac{1}{x-4} & -\frac{1}{x-5} & -\frac{1}{x-6} \\ -\frac{1}{x-4} & -\frac{1}{x-5} & -\frac{1}{x-6} & -\frac{1}{x-7} \\ -\frac{1}{x-5} & -\frac{1}{x-6} & -\frac{1}{x-7} & -\frac{1}{x-8} \end{bmatrix}$$

<div align="right">Type: Matrix Fraction Polynomial Integer</div>

Let vm denote the three by three Vandermonde matrix.

```
vm := matrix [ [1,1,1], [x,y,z], [x*x,y*y,z*z] ]
```

$$\begin{bmatrix} 1 & 1 & 1 \\ x & y & z \\ x^2 & y^2 & z^2 \end{bmatrix}$$

<div align="right">Type: Matrix Polynomial Integer</div>

Use this syntax to extract an entry in the matrix.

```
vm(3,3)
```

$$z^2$$

$$\text{Type: Polynomial Integer}$$

You can also pull out a **row** or a **column**.

`column(vm,2)`

$$\left[1, y, y^2\right]$$

$$\text{Type: Vector Polynomial Integer}$$

You can do arithmetic.

`vm * vm`

$$\begin{bmatrix} x^2 + x + 1 & y^2 + y + 1 & z^2 + z + 1 \\ x^2\,z + x\,y + x & y^2\,z + y^2 + x & z^3 + y\,z + x \\ x^2\,z^2 + x\,y^2 + x^2 & y^2\,z^2 + y^3 + x^2 & z^4 + y^2\,z + x^2 \end{bmatrix}$$

$$\text{Type: Matrix Polynomial Integer}$$

You can perform operations such as **transpose**, **trace**, and **determinant**.

`factor determinant vm`

$$(y - x)\ (z - y)\ (z - x)$$

$$\text{Type: Factored Polynomial Integer}$$

1.7   Writing Your Own Functions

Axiom provides you with a very large library of predefined operations and objects to compute with. You can use the Axiom library of constructors to create new objects dynamically of quite arbitrary complexity. For example, you can make lists of matrices of fractions of polynomials with complex floating point numbers as coefficients. Moreover, the library provides a wealth of operations that allow you to create and manipulate these objects.

For many applications, you need to interact with the interpreter and write some Axiom programs to tackle your application. Axiom allows you to write functions interactively, thereby effectively extending the system library. Here we give a few simple examples, leaving the details to Chapter 6 on page 231.

We begin by looking at several ways that you can define the "factorial" function in Axiom. The first way is to give a piece-wise definition of the function. This

method is best for a general recurrence relation since the pieces are gathered together and compiled into an efficient iterative function. Furthermore, enough previously computed values are automatically saved so that a subsequent call to the function can pick up from where it left off.

Define the value of **fact** at 0.

```
fact(0) == 1
```

                                                    Type: Void

Define the value of **fact**(n) for general n.

```
fact(n) == n*fact(n-1)
```

                                                    Type: Void

Ask for the value at 50. The resulting function created by Axiom computes the value by iteration.

```
fact(50)
```

   Compiling function fact with type Integer -> Integer
   Compiling function fact as a recurrence relation.

   30414093201713378043612608166064768844377641568960512000000000000

                                            Type: PositiveInteger

A second definition uses an `if-then-else` and recursion.

```
fac(n) == if n < 3 then n else n * fac(n - 1)
```

                                                    Type: Void

This function is less efficient than the previous version since each iteration involves a recursive function call.

```
fac(50)
```

   30414093201713378043612608166064768844377641568960512000000000000

                                               Type: PositiveInteger

A third version directly uses iteration.

```
fa(n) == (a := 1; for i in 2..n repeat a := a*i; a)
```

                                                           Type: Void

This is the least space-consumptive version.

```
fa(50)
```

```
   Compiling function fac with type Integer -> Integer
```

   30414093201713378043612608166064768844377641568960512000000000000

                                               Type: PositiveInteger

A final version appears to construct a large list and then reduces over it with
multiplication.

```
f(n) == reduce(*,[i for i in 2..n])
```

                                                           Type: Void

In fact, the resulting computation is optimized into an efficient iteration loop
equivalent to that of the third version.

```
f(50)
```

```
Compiling function f with type
   PositiveInteger -> PositiveInteger
```

   30414093201713378043612608166064768844377641568960512000000000000

                                               Type: PositiveInteger

The library version uses an algorithm that is different from the four above
because it highly optimizes the recurrence relation definition of **factorial**.

```
factorial(50)
```

304140932017133780436126081660647688443776415689605120000000000000

<div align="right">Type: PositiveInteger</div>

You are not limited to one-line functions in Axiom. If you place your function definitions in **.input** files (see 4.1 on page 183), you can have multi-line functions that use indentation for grouping.

Given n elements, **diagonalMatrix** creates an n by n matrix with those elements down the diagonal. This function uses a permutation matrix that interchanges the ith and jth rows of a matrix by which it is right-multiplied.

This function definition shows a style of definition that can be used in **.input** files. Indentation is used to create *blocks*: sequences of expressions that are evaluated in sequence except as modified by control statements such as `if-then-else` and `return`.

```
permMat(n, i, j) ==
  m := diagonalMatrix
    [(if i = k or j = k then 0 else 1)
      for k in 1..n]
  m(i,j) := 1
  m(j,i) := 1
  m
```

This creates a four by four matrix that interchanges the second and third rows.

```
p := permMat(4,2,3)
```

```
   Compiling function permMat with type (PositiveInteger,
      PositiveInteger,PositiveInteger) -> Matrix Integer
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

<div align="right">Type: Matrix Integer</div>

Create an example matrix to permute.

```
m := matrix [ [4*i + j for j in 1..4] for i in 0..3]
```

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

<div align="right">Type: Matrix Integer</div>

Interchange the second and third rows of m.

`permMat(4,2,3) * m`

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 9 & 10 & 11 & 12 \\ 5 & 6 & 7 & 8 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

<div align="right">Type: Matrix Integer</div>

A function can also be passed as an argument to another function, which then applies the function or passes it off to some other function that does. You often have to declare the type of a function that has functional arguments.

This declares **t** to be a two-argument function that returns a `Float`. The first argument is a function that takes one `Float` argument and returns a `Float`.

`t :  (Float -> Float, Float) -> Float`

<div align="right">Type: Void</div>

This is the definition of **t**.

`t(fun, x) == fun(x)**2 + sin(x)**2`

<div align="right">Type: Void</div>

We have not defined a **cos** in the workspace. The one from the Axiom library will do.

`t(cos, 5.2058)`

<div align="center">1.0</div>

<div align="right">Type: Float</div>

Here we define our own (user-defined) function.

`cosinv(y) == cos(1/y)`

$$\text{Type: Void}$$

Pass this function as an argument to **t**.

```
t(cosinv, 5.2058)
```

$$1.7392237241\ 8005164925\ 4147684772\ 932520785$$

$$\text{Type: Float}$$

Axiom also has pattern matching capabilities for simplification of expressions and for defining new functions by rules. For example, suppose that you want to apply regularly a transformation that groups together products of radicals:

$$\sqrt{a}\sqrt{b} \mapsto \sqrt{ab}, \quad (\forall a)(\forall b)$$

Note that such a transformation is not generally correct. Axiom never uses it automatically.

Give this rule the name **groupSqrt**.

```
groupSqrt := rule(sqrt(a) * sqrt(b) == sqrt(a*b))
```

$$\%C\ \sqrt{a}\ \sqrt{b} == \%C\ \sqrt{a\ b}$$

$$\text{Type: RewriteRule(Integer,Integer,Expression Integer)}$$

Here is a test expression.

```
a := (sqrt(x) + sqrt(y) + sqrt(z))**4
```

$$\left((4\ z + 4\ y + 12\ x)\ \sqrt{y} + (4\ z + 12\ y + 4\ x)\ \sqrt{x}\right)\ \sqrt{z} +$$

$$(12\ z + 4\ y + 4\ x)\ \sqrt{x}\ \sqrt{y} + z^2 + (6\ y + 6\ x)\ z + y^2 + 6\ x\ y + x^2$$

$$\text{Type: Expression Integer}$$

The rule **groupSqrt** successfully simplifies the expression.

```
groupSqrt a
```

$$(4\ z + 4\ y + 12\ x)\ \sqrt{y\ z} + (4\ z + 12\ y + 4\ x)\ \sqrt{x\ z} +$$

$$(12\ z + 4\ y + 4\ x)\ \sqrt{x\ y} + z^2 + (6\ y + 6\ x)\ z + y^2 + 6\ x\ y + x^2$$

$$\text{Type: Expression Integer}$$

1.8   Polynomials

Polynomials are the commonly used algebraic types in symbolic computation.
Interactive users of Axiom generally only see one type of polynomial: `Polynomial(R)`.
This type represents polynomials in any number of unspecified variables over a
particular coefficient domain R. This type represents its coefficients *sparsely*:
only terms with non-zero coefficients are represented.

In building applications, many other kinds of polynomial representations are
useful. Polynomials may have one variable or multiple variables, the variables
can be named or unnamed, the coefficients can be stored sparsely or densely. So-
called "distributed multivariate polynomials" store polynomials as coefficients
paired with vectors of exponents. This type is particularly efficient for use in
algorithms for solving systems of non-linear polynomial equations.

The polynomial constructor most familiar to the interactive user is `Polynomial`.

```
(x**2 - x*y**3 +3*y)**2
```

$$x^2 \; y^6 - 6 \; x \; y^4 - 2 \; x^3 \; y^3 + 9 \; y^2 + 6 \; x^2 \; y + x^4$$

```
                                    Type:  Polynomial Integer
```

If you wish to restrict the variables used, `UnivariatePolynomial` provides poly-
nomials in one variable.

```
p:  UP(x,INT) := (3*x-1)**2 * (2*x + 8)
```

$$18 \; x^3 + 60 \; x^2 - 46 \; x + 8$$

```
                            Type:  UnivariatePolynomial(x,Integer)
```

The constructor `MultivariatePolynomial` provides polynomials in one or more
specified variables.

```
m:  MPOLY([x,y],INT) := (x**2-x*y**3+3*y)**2
```

$$x^4 - 2 \; y^3 \; x^3 + \left(y^6 + 6 \; y \right) \; x^2 - 6 \; y^4 \; x + 9 \; y^2$$

```
                        Type:  MultivariatePolynomial([x,y],Integer)
```

You can change the way the polynomial appears by modifying the variable
ordering in the explicit list.

```
m ::  MPOLY([y,x],INT)
```

$$x^2 \ y^6 - 6 \ x \ y^4 - 2 \ x^3 \ y^3 + 9 \ y^2 + 6 \ x^2 \ y + x^4$$

<div align="center">Type: MultivariatePolynomial([y,x],Integer)</div>

The constructor `DistributedMultivariatePolynomial` provides polynomials
in one or more specified variables with the monomials ordered lexicographically.

```
m ::  DMP([y,x],INT)
```

$$y^6 \ x^2 - 6 \ y^4 \ x - 2 \ y^3 \ x^3 + 9 \ y^2 + 6 \ y \ x^2 + x^4$$

<div align="center">Type: DistributedMultivariatePolynomial([y,x],Integer)</div>

The constructor `HomogeneousDistributedMultivariatePolynomial` is similar
except that the monomials are ordered by total order refined by reverse lexico-
graphic order.

```
m ::  HDMP([y,x],INT)
```

$$y^6 \ x^2 - 2 \ y^3 \ x^3 - 6 \ y^4 \ x + x^4 + 6 \ y \ x^2 + 9 \ y^2$$

<div align="right">Type:</div>
<div align="center">HomogeneousDistributedMultivariatePolynomial([y,x],Integer)</div>

More generally, the domain constructor `GeneralDistributedMultivariatePolynomial`
allows the user to provide an arbitrary predicate to define his own term ordering.
These last three constructors are typically used in Gröbner basis applications
and when a flat (that is, non-recursive) display is wanted and the term ordering
is critical for controlling the computation.

1.9 Limits

Axiom's **limit** function is usually used to evaluate limits of quotients where the
numerator and denominator both tend to zero or both tend to infinity. To find
the limit of an expression f as a real variable x tends to a limit value a, enter
`limit(f, x=a)`. Use **complexLimit** if the variable is complex. Additional
information and examples of limits are in Section 8.6 on page 373.

You can take limits of functions with parameters.

```
g := csc(a*x) / csch(b*x)
```

$$\frac{\csc{(a \ x)}}{\operatorname{csch}{(b \ x)}}$$

<div align="right">Type: Expression Integer</div>

As you can see, the limit is expressed in terms of the parameters.

`limit(g,x=0)`

$$\frac{b}{a}$$

<div align="center">Type: Union(OrderedCompletion Expression Integer,...)</div>

A variable may also approach plus or minus infinity:

`h := (1 + k/x)**x`

$$\frac{x+k}{x}^{x}$$

<div align="right">Type: Expression Integer</div>

Use %plusInfinity and %minusInfinity to denote ∞ and $-\infty$.

`limit(h,x=%plusInfinity)`

$$e^{k}$$

<div align="center">Type: Union(OrderedCompletion Expression Integer,...)</div>

A function can be defined on both sides of a particular value, but may tend to different limits as its variable approaches that value from the left and from the right.

`limit(sqrt(y**2)/y,y = 0)`

$$[leftHandLimit = -1, rightHandLimit = 1]$$

<div align="center">Type: Union(Record(leftHandLimit: Union(OrderedCompletion
Expression Integer,"failed"),rightHandLimit:
Union(OrderedCompletion Expression Integer,"failed")),...)</div>

As x approaches 0 along the real axis, `exp(-1/x**2)` tends to 0.

`limit(exp(-1/x**2),x = 0)`

$$0$$

Type: Union(OrderedCompletion Expression Integer,...)

However, if x is allowed to approach 0 along any path in the complex plane, the limiting value of `exp(-1/x**2)` depends on the path taken because the function has an essential singularity at $x = 0$. This is reflected in the error message returned by the function.

`complexLimit(exp(-1/x**2),x = 0)`

$$\text{"failed"}$$

Type: Union("failed",...)

1.10 Series

Axiom also provides power series. By default, Axiom tries to compute and display the first ten elements of a series. Use `)set streams calculate` to change the default value to something else. For the purposes of this document, we have used this system command to display fewer than ten terms. For more information about working with series, see 8.9 on page 383.

You can convert a functional expression to a power series by using the operation **series**. In this example, `sin(a*x)` is expanded in powers of $(x - 0)$, that is, in powers of x.

`series(sin(a*x),x = 0)`

$$a\ x - \frac{a^3}{6}\ x^3 + \frac{a^5}{120}\ x^5 - \frac{a^7}{5040}\ x^7 + \frac{a^9}{362880}\ x^9 - \frac{a^{11}}{39916800}\ x^{11} + O\left(x^{12}\right)$$

Type: UnivariatePuiseuxSeries(Expression Integer,x,0)

This expression expands `sin(a*x)` in powers of `(x - %pi/4)`.

`series(sin(a*x),x = %pi/4)`

$$\sin\left(\frac{a\ \pi}{4}\right) + a\ \cos\left(\frac{a\ \pi}{4}\right)\ \left(x - \frac{\pi}{4}\right) -$$

$$\frac{a^2\ \sin\left(\frac{a\ \pi}{4}\right)}{2}\ \left(x - \frac{\pi}{4}\right)^2 - \frac{a^3\ \cos\left(\frac{a\ \pi}{4}\right)}{6}\ \left(x - \frac{\pi}{4}\right)^3 +$$

$$\frac{a^4\ \sin\left(\frac{a\ \pi}{4}\right)}{24}\ \left(x - \frac{\pi}{4}\right)^4 + \frac{a^5\ \cos\left(\frac{a\ \pi}{4}\right)}{120}\ \left(x - \frac{\pi}{4}\right)^5 -$$

$$\frac{a^6 \sin\left(\frac{a\ \pi}{4}\right)}{720} \left(x - \frac{\pi}{4}\right)^6 - \frac{a^7 \cos\left(\frac{a\ \pi}{4}\right)}{5040} \left(x - \frac{\pi}{4}\right)^7 +$$

$$\frac{a^8 \sin\left(\frac{a\ \pi}{4}\right)}{40320} \left(x - \frac{\pi}{4}\right)^8 + \frac{a^9 \cos\left(\frac{a\ \pi}{4}\right)}{362880} \left(x - \frac{\pi}{4}\right)^9 -$$

$$\frac{a^{10} \sin\left(\frac{a\ \pi}{4}\right)}{3628800} \left(x - \frac{\pi}{4}\right)^{10} + O\left(\left(x - \frac{\pi}{4}\right)^{11}\right)$$

        Type: UnivariatePuiseuxSeries(Expression Integer,x,pi/4)

Axiom provides *Puiseux series:* series with rational number exponents. The first argument to **series** is an in-place function that computes the n-th coefficient. (Recall that the "+->" is an infix operator meaning "maps to.")

```
series(n +-> (-1)**((3*n - 4)/6)/factorial(n - 1/3),x=0,4/3..,2)
```

$$x^{\frac{4}{3}} - \frac{1}{6} x^{\frac{10}{3}} + O\left(x^5\right)$$

        Type: UnivariatePuiseuxSeries(Expression Integer,x,0)

Once you have created a power series, you can perform arithmetic operations on that series. We compute the Taylor expansion of $1/(1 - x)$.

```
f := series(1/(1-x),x = 0)
```

$$1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9 + x^{10} + O\left(x^{11}\right)$$

        Type: UnivariatePuiseuxSeries(Expression Integer,x,0)

Compute the square of the series.

```
f ** 2
```

$$1 + 2\ x + 3\ x^2 + 4\ x^3 + 5\ x^4 + 6\ x^5 + 7\ x^6 + 8\ x^7 + 9\ x^8 + 10\ x^9 + 11\ x^{10} + O\left(x^{11}\right)$$

        Type: UnivariatePuiseuxSeries(Expression Integer,x,0)

The usual elementary functions (**log**, **exp**, trigonometric functions, and so on) are defined for power series.

```
f := series(1/(1-x),x = 0)
```

$$1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9 + x^{10} + O\left(x^{11}\right)$$

Type: UnivariatePuiseuxSeries(Expression Integer,x,0)

g := log(f)

$$x + \tfrac{1}{2}\,x^2 + \tfrac{1}{3}\,x^3 + \tfrac{1}{4}\,x^4 + \tfrac{1}{5}\,x^5 + \tfrac{1}{6}\,x^6 + \tfrac{1}{7}\,x^7 +$$

$$\frac{1}{8}\,x^8 + \frac{1}{9}\,x^9 + \frac{1}{10}\,x^{10} + \frac{1}{11}\,x^{11} + O\left(x^{12}\right)$$

Type: UnivariatePuiseuxSeries(Expression Integer,x,0)

exp(g)

$$1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9 + x^{10} + O\left(x^{11}\right)$$

Type: UnivariatePuiseuxSeries(Expression Integer,x,0)

Here is a way to obtain numerical approximations of e from the Taylor series expansion of **exp**(x). First create the desired Taylor expansion.

f := taylor(exp(x))

$$1 + x + \frac{1}{2}\,x^2 + \frac{1}{6}\,x^3 + \frac{1}{24}\,x^4 + \frac{1}{120}\,x^5 + \frac{1}{720}\,x^6 +$$

$$\frac{1}{5040}\,x^7 + \frac{1}{40320}\,x^8 + \frac{1}{362880}\,x^9 + \frac{1}{3628800}\,x^{10} + O\left(x^{11}\right)$$

Type: UnivariateTaylorSeries(Expression Integer,x,0)

Evaluate the series at the value 1.0. As you see, you get a sequence of partial sums.

eval(f,1.0)

$$[1.0, 2.0, 2.5, 2.6666666666666666667,$$

$$2.7083333333333333333, 2.7166666666666666667,$$

$$2.7180555555555555556, 2.718253968253968254,$$

$$2.7182787698412698413, 2.7182815255731922399, \ldots \quad]$$

Type: Stream Expression Float

1.11   Derivatives

Use the Axiom function **D** to differentiate an expression.

To find the derivative of an expression f with respect to a variable x, enter **D**(f, x).

```
f := exp exp x
```

$$e^{e^x}$$

<div align="right">Type: Expression Integer</div>

```
D(f, x)
```

$$e^x \; e^{e^x}$$

<div align="right">Type: Expression Integer</div>

An optional third argument n in **D** asks Axiom for the n-th derivative of f. This finds the fourth derivative of f with respect to x.

```
D(f, x, 4)
```

$$\left(e^{x4} + 6 \; e^{x3} + 7 \; e^{x2} + e^x\right) e^{e^x}$$

<div align="right">Type: Expression Integer</div>

You can also compute partial derivatives by specifying the order of differentiation.

```
g := sin(x**2 + y)
```

$$\sin\left(y + x^2\right)$$

<div align="right">Type: Expression Integer</div>

```
D(g, y)
```

$$\cos\left(y + x^2\right)$$

<div align="right">Type: Expression Integer</div>

```
D(g, [y, y, x, x])
```

$$4\ x^2 \sin\left(y + x^2\right) - 2\ \cos\left(y + x^2\right)$$

<div align="right">Type: Expression Integer</div>

Axiom can manipulate the derivatives (partial and iterated) of expressions involving formal operators. All the dependencies must be explicit.

This returns 0 since F (so far) does not explicitly depend on x.

```
D(F,x)
```

$$0$$

<div align="right">Type: Polynomial Integer</div>

Suppose that we have F a function of x, y, and z, where x and y are themselves functions of z.

Start by declaring that F, x, and y are operators.

```
F := operator 'F; x := operator 'x; y := operator 'y
```

$$y$$

<div align="right">Type: BasicOperator</div>

You can use F, x, and y in expressions.

```
a := F(x z, y z, z**2) + x y(z+1)
```

$$x\left(y\left(z + 1\right)\right) + F\left(x\left(z\right), y\left(z\right), z^2\right)$$

<div align="right">Type: Expression Integer</div>

Differentiate formally with respect to z. The formal derivatives appearing in *dadz* are not just formal symbols, but do represent the derivatives of x, y, and F.

```
dadz := D(a, z)
```

$$2\ z\ F_{,3}\left(x\left(z\right), y\left(z\right), z^2\right) + y\text{'}\left(z\right)\ F_{,2}\left(x\left(z\right), y\left(z\right), z^2\right) +$$

$$x\text{'}\left(z\right)\ F_{,1}\left(x\left(z\right), y\left(z\right), z^2\right) + x\text{'}\left(y\left(z + 1\right)\right)\ y\text{'}\left(z + 1\right)$$

<div align="right">Type: Expression Integer</div>

You can evaluate the above for particular functional values of F, x, and y. If $x(z)$ is **exp**(z) and $y(z)$ is **log**(z+1), then evaluates `dadz`.

```
eval(eval(dadz, 'x, z +-> exp z), 'y, z +-> log(z+1))
```

$$\frac{\begin{pmatrix} \left(2\ z^2 + 2\ z\right)\ F_{,3}\left(e^z, \log\left(z+1\right), z^2\right)+ \\[2mm] F_{,2}\left(e^z, \log\left(z+1\right), z^2\right)+ \\[2mm] \left(z+1\right)\ e^z\ F_{,1}\left(e^z, \log\left(z+1\right), z^2\right) + z + 1 \end{pmatrix}}{z+1}$$

<div align="right">Type: Expression Integer</div>

You obtain the same result by first evaluating a and then differentiating.

```
eval(eval(a, 'x, z +-> exp z), 'y, z +-> log(z+1))
```

$$F\left(e^z, \log\left(z+1\right), z^2\right) + z + 2$$

<div align="right">Type: Expression Integer</div>

```
D(%, z)
```

$$\frac{\begin{pmatrix} \left(2\ z^2 + 2\ z\right)\ F_{,3}\left(e^z, \log\left(z+1\right), z^2\right)+ \\[2mm] F_{,2}\left(e^z, \log\left(z+1\right), z^2\right)+ \\[2mm] \left(z+1\right)\ e^z\ F_{,1}\left(e^z, \log\left(z+1\right), z^2\right) + z + 1 \end{pmatrix}}{z+1}$$

<div align="right">Type: Expression Integer</div>

1.12   Integration

Axiom has extensive library facilities for integration.

The first example is the integration of a fraction with denominator that factors into a quadratic and a quartic irreducible polynomial. The usual partial fraction approach used by most other computer algebra systems either fails or introduces expensive unneeded algebraic numbers.

We use a factorization-free algorithm.

```
integrate((x**2+2*x+1)/((x+1)**6+1),x)
```

$$\frac{\arctan\left(x^3 + 3\ x^2 + 3\ x + 1\right)}{3}$$

<div align="right">Type: Union(Expression Integer,...)</div>

When real parameters are present, the form of the integral can depend on the signs of some expressions.

Rather than query the user or make sign assumptions, Axiom returns all possible answers.

```
integrate(1/(x**2 + a),x)
```

$$\left[\frac{\log\left(\frac{(x^2-a)\ \sqrt{-a}+2\ a\ x}{x^2+a}\right)}{2\ \sqrt{-a}}, \frac{\arctan\left(\frac{x\ \sqrt{a}}{a}\right)}{\sqrt{a}}\right]$$

<div align="right">Type: Union(List Expression Integer,...)</div>

The **integrate** operation generally assumes that all parameters are real. The only exception is when the integrand has complex valued quantities.

If the parameter is complex instead of real, then the notion of sign is undefined and there is a unique answer. You can request this answer by "prepending" the word "complex" to the command name:

```
complexIntegrate(1/(x**2 + a),x)
```

$$\frac{\log\left(\frac{x\ \sqrt{-a}+a}{\sqrt{-a}}\right) - \log\left(\frac{x\ \sqrt{-a}-a}{\sqrt{-a}}\right)}{2\ \sqrt{-a}}$$

<div align="right">Type: Expression Integer</div>

The following two examples illustrate the limitations of table-based approaches. The two integrands are very similar, but the answer to one of them requires the addition of two new algebraic numbers.

This one is the easy one. The next one looks very similar but the answer is much more complicated.

```
integrate(x**3 / (a+b*x)**(1/3),x)
```

$$\frac{\left(120\ b^3\ x^3 - 135\ a\ b^2\ x^2 + 162\ a^2\ b\ x - 243\ a^3\right)\ \sqrt[3]{b\ x + a}^2}{440\ b^4}$$

$$\text{Type: Union(Expression Integer,...)}$$

Only an algorithmic approach is guaranteed to find what new constants must be added in order to find a solution.

```
integrate(1 / (x**3 * (a+b*x)**(1/3)),x)
```

$$\frac{\left(\begin{array}{l}-2\ b^2\ x^2\ \sqrt{3}\ \log\left(\sqrt[3]{a}\ \sqrt[3]{b\ x+a}^2+\sqrt[3]{a}^2\ \sqrt[3]{b\ x+a}+a\right)+\\[2ex] 4\ b^2\ x^2\ \sqrt{3}\ \log\left(\sqrt[3]{a}^2\ \sqrt[3]{b\ x+a}-a\right)+\\[2ex] 12\ b^2\ x^2\ \arctan\left(\dfrac{2\ \sqrt{3}\ \sqrt[3]{a}^2\ \sqrt[3]{b\ x+a}+a\ \sqrt{3}}{3\ a}\right)+\\[2ex] (12\ b\ x-9\ a)\ \sqrt{3}\ \sqrt[3]{a}\ \sqrt[3]{b\ x+a}^2\end{array}\right)}{18\ a^2\ x^2\ \sqrt{3}\ \sqrt[3]{a}}$$

$$\text{Type: Union(Expression Integer,...)}$$

Some computer algebra systems use heuristics or table-driven approaches to integration. When these systems cannot determine the answer to an integration problem, they reply "I don't know." Axiom uses an algorithm which is a *decision procedure* for integration. If Axiom returns the original integral that conclusively proves that an integral cannot be expressed in terms of elementary functions.

When Axiom returns an integral sign, it has proved that no answer exists as an elementary function.

```
integrate(log(1 + sqrt(a*x + b)) / x,x)
```

$$\int^{x}\frac{\log\left(\sqrt{b+\%Q\ a}+1\right)}{\%Q}\ d\%Q$$

$$\text{Type: Union(Expression Integer,...)}$$

Axiom can handle complicated mixed functions much beyond what you can find in tables.

Whenever possible, Axiom tries to express the answer using the functions present in the integrand.

```
integrate((sinh(1+sqrt(x+b))+2*sqrt(x+b)) / (sqrt(x+b) * (x +
cosh(1+sqrt(x + b)))), x)
```

$$2 \log \left(\frac{-2 \cosh \left(\sqrt{x+b}+1 \right) - 2\ x}{\sinh \left(\sqrt{x+b}+1 \right) - \cosh \left(\sqrt{x+b}+1 \right)} \right) - 2\ \sqrt{x+b}$$

<div align="right">Type: Union(Expression Integer,...)</div>

A strong structure-checking algorithm in Axiom finds hidden algebraic relationships between functions.

```
integrate(tan(atan(x)/3),x)
```

$$\frac{\left(8 \log \left(3 \tan \left(\frac{\arctan(x)}{3} \right)^2 - 1 \right) - 3 \tan \left(\frac{\arctan(x)}{3} \right)^2 + 18\ x \tan \left(\frac{\arctan(x)}{3} \right) \right)}{18}$$

<div align="right">Type: Union(Expression Integer,...)</div>

The discovery of this algebraic relationship is necessary for correct integration of this function. Here are the details:

1. If $x = \tan t$ and $g = \tan(t/3)$ then the following algebraic relation is true:

$$g^3 - 3xg^2 - 3g + x = 0$$

2. Integrate g using this algebraic relation; this produces:

$$\frac{(24g^2 - 8) \log(3g^2 - 1) + (81x^2 + 24)g^2 + 72xg - 27x^2 - 16}{54g^2 - 18}$$

3. Rationalize the denominator, producing:

$$\frac{8 \log(3g^2 - 1) - 3g^2 + 18xg + 16}{18}$$

Replace g by the initial definition $g = \tan(\arctan(x)/3)$ to produce the final result.

This is an example of a mixed function where the algebraic layer is over the transcendental one.

```
integrate((x + 1) / (x*(x + log x) ** (3/2)), x)
```

$$-\frac{2 \sqrt{\log(x) + x}}{\log(x) + x}$$

While incomplete for non-elementary functions, Axiom can handle some of them.

```
integrate(exp(-x**2) * erf(x) / (erf(x)**3 - erf(x)**2 - erf(x) +
1),x)
```

$$\frac{(\mathrm{erf}\,(x) - 1)\ \sqrt{\pi}\ \log\left(\frac{\mathrm{erf}(x)-1}{\mathrm{erf}(x)+1}\right) - 2\ \sqrt{\pi}}{8\ \mathrm{erf}\,(x) - 8}$$

Type: Union(Expression Integer,...)

More examples of Axiom's integration capabilities are discussed in Section 8.8 on page 379.

1.13   Differential Equations

The general approach used in integration also carries over to the solution of linear differential equations.

Let's solve some differential equations. Let y be the unknown function in terms of x.

```
y := operator 'y
```

$$y$$

Type: BasicOperator

Here we solve a third order equation with polynomial coefficients.

```
deq := x**3 * D(y x, x, 3) + x**2 * D(y x, x, 2) - 2 * x * D(y x,
x) + 2 * y x = 2 * x**4
```

$$x^3\ y'''\,(x) + x^2\ y''\,(x) - 2\ x\ y'\,(x) + 2\ y\,(x) = 2\ x^4$$

Type: Equation Expression Integer

```
solve(deq, y, x)
```

$$\left[particular = \tfrac{x^5 - 10\ x^3 + 20\ x^2 + 4}{15\ x},\right.$$

$$\left.basis = \left[\frac{2\ x^3 - 3\ x^2 + 1}{x}, \frac{x^3 - 1}{x}, \frac{x^3 - 3\ x^2 - 1}{x}\right]\right]$$

```
  Type:  Union(Record(particular:  Expression Integer,basis:  List
                                     Expression Integer),...)
```

Here we find all the algebraic function solutions of the equation.

```
deq := (x**2 + 1) * D(y x, x, 2) + 3 * x * D(y x, x) + y x = 0
```

$$\left(x^2 + 1\right) y'' (x) + 3 x y' (x) + y (x) = 0$$

```
                    Type:  Equation Expression Integer
```

```
solve(deq, y, x)
```

$$\left[particular = 0, basis = \left[\frac{1}{\sqrt{x^2 + 1}}, \frac{\log \left(\sqrt{x^2 + 1} - x \right)}{\sqrt{x^2 + 1}} \right] \right]$$

```
  Type:  Union(Record(particular:  Expression Integer,basis:  List
                                     Expression Integer),...)
```

Coefficients of differential equations can come from arbitrary constant fields. For example, coefficients can contain algebraic numbers.

This example has solutions whose logarithmic derivative is an algebraic function of degree two.

```
eq := 2*x**3 * D(y x,x,2) + 3*x**2 * D(y x,x) - 2 * y x
```

$$2 x^3 y'' (x) + 3 x^2 y' (x) - 2 y (x)$$

```
                            Type:  Expression Integer
```

```
solve(eq,y,x).basis
```

$$\left[e^{\left(-\frac{2}{\sqrt{x}} \right)}, e^{\frac{2}{\sqrt{x}}} \right]$$

```
                        Type:  List Expression Integer
```

Here's another differential equation to solve.

```
deq := D(y x, x) = y(x) / (x + y(x) * log y x)
```

$$y'(x) = \frac{y(x)}{y(x)\,\log(y(x)) + x}$$

<div align="center">Type: Equation Expression Integer</div>

```
solve(deq, y, x)
```

$$\frac{y(x)\,\log(y(x))^2 - 2\,x}{2\,y(x)}$$

<div align="center">Type: Union(Expression Integer,...)</div>

Rather than attempting to get a closed form solution of a differential equation, you instead might want to find an approximate solution in the form of a series.

Let's solve a system of nonlinear first order equations and get a solution in power series. Tell Axiom that x is also an operator.

```
x := operator 'x
```

$$x$$

<div align="center">Type: BasicOperator</div>

Here are the two equations forming our system.

```
eq1 := D(x(t), t) = 1 + x(t)**2
```

$$x'(t) = x(t)^2 + 1$$

<div align="center">Type: Equation Expression Integer</div>

```
eq2 := D(y(t), t) = x(t) * y(t)
```

$$y'(t) = x(t)\,y(t)$$

<div align="center">Type: Equation Expression Integer</div>

We can solve the system around $t = 0$ with the initial conditions $x(0) = 0$ and $y(0) = 1$. Notice that since we give the unknowns in the order $[x, y]$, the answer is a list of two series in the order [series for $x(t)$, series for $y(t)$].

```
seriesSolve([eq2, eq1], [x, y], t = 0, [y(0) = 1, x(0) = 0])
```

$$\left[t + \frac{1}{3} \, t^3 + \frac{2}{15} \, t^5 + \frac{17}{315} \, t^7 + \frac{62}{2835} \, t^9 + O\left(t^{11}\right), \right.$$

$$\left. 1 + \frac{1}{2} \, t^2 + \frac{5}{24} \, t^4 + \frac{61}{720} \, t^6 + \frac{277}{8064} \, t^8 + \frac{50521}{3628800} \, t^{10} + O\left(t^{11}\right) \right]$$

```
Type:  List UnivariateTaylorSeries(Expression Integer,t,0)
```

1.14 Solution of Equations

Axiom also has state-of-the-art algorithms for the solution of systems of polynomial equations. When the number of equations and unknowns is the same, and you have no symbolic coefficients, you can use **solve** for real roots and **complexSolve** for complex roots. In each case, you tell Axiom how accurate you want your result to be. All operations in the *solve* family return answers in the form of a list of solution sets, where each solution set is a list of equations.

A system of two equations involving a symbolic parameter t.

```
S(t) == [x**2-2*y**2 - t,x*y-y-5*x + 5]
```

```
                                                      Type:  Void
```

Find the real roots of $S(19)$ with rational arithmetic, correct to within $1/10^{20}$.

```
solve(S(19),1/10**20)
```

$$\left[\left[y = 5, x = -\frac{2451682632253093442511}{295147905179352825856}\right],\right.$$

$$\left.\left[y = 5, x = \frac{2451682632253093442511}{295147905179352825856}\right]\right]$$

```
        Type:  List List Equation Polynomial Fraction Integer
```

Find the complex roots of $S(19)$ with floating point coefficients to 20 digits accuracy in the mantissa.

```
complexSolve(S(19),10.e-20)
```

$$[[y = 5.0, x = 8.3066238629180748526],$$
$$[y = 5.0, x = -8.3066238629180748526],$$
$$[y = -3.0 \; i, x = 1.0], [y = 3.0 \; i, x = 1.0]]$$

If a system of equations has symbolic coefficients and you want a solution in radicals, try **radicalSolve**.

```
radicalSolve(S(a),[x,y])
```

$$\left[\left[x = -\sqrt{a+50}, y = 5\right], \left[x = \sqrt{a+50}, y = 5\right],\right.$$
$$\left.\left[x = 1, y = \sqrt{\frac{-a+1}{2}}\right], \left[x = 1, y = -\sqrt{\frac{-a+1}{2}}\right]\right]$$

Type: List List Equation Expression Integer

For systems of equations with symbolic coefficients, you can apply **solve**, listing the variables that you want Axiom to solve for. For polynomial equations, a solution cannot usually be expressed solely in terms of the other variables. Instead, the solution is presented as a "triangular" system of equations, where each polynomial has coefficients involving only the succeeding variables. This is analogous to converting a linear system of equations to "triangular form".

A system of three equations in five variables.

```
eqns := [x**2 - y + z,x**2*z + x**4 - b*y, y**2 *z - a - b*x]
```

$$\left[z - y + x^2, x^2\ z - b\ y + x^4, y^2\ z - b\ x - a\right]$$

Type: List Polynomial Integer

Solve the system for unknowns $[x, y, z]$, reducing the solution to triangular form.

```
solve(eqns,[x,y,z])
```

$$\left[\left[x = -\frac{a}{b}, y = 0, z = -\frac{a^2}{b^2}\right],\right.$$

$$\left[x = \frac{z^3+2\ b\ z^2+b^2\ z-a}{b}, y = z + b,\right.$$

$$z^6 + 4\ b\ z^5 + 6\ b^2\ z^4 + \left(4\ b^3 - 2\ a\right) z^3 + \left(b^4 - 4\ a\ b\right) z^2 -$$

$$\left. 2\ a\ b^2\ z - b^3 + a^2 = 0\right]$$

Type: List List Equation Fraction Polynomial Integer

1.15 System Commands

We conclude our tour of Axiom with a brief discussion of *system commands*. System commands are special statements that start with a closing parenthesis ()). They are used to control or display your Axiom environment, start the HyperDoc system, issue operating system commands and leave Axiom. For example, `)system` is used to issue commands to the operating system from Axiom. Here is a brief description of some of these commands. For more information on specific commands, see Appendix A on page 987.

Perhaps the most important user command is the `)clear all` command that initializes your environment. Every section and subsection in this document has an invisible `)clear all` that is read prior to the examples given in the section. `)clear all` gives you a fresh, empty environment with no user variables defined and the step number reset to 1. The `)clear` command can also be used to selectively clear values and properties of system variables.

Another useful system command is `)read`. A preferred way to develop an application in Axiom is to put your interactive commands into a file, say **my.input** file. To get Axiom to read this file, you use the system command `)read my.input`. If you need to make changes to your approach or definitions, go into your favorite editor, change **my.input**, then `)read my.input` again.

Other system commands include: `)history`, to display previous input and/or output lines; `)display`, to display properties and values of workspace variables; and `)what`.

Issue `)what` to get a list of Axiom objects that contain a given substring in their name.

```
)what operations integrate
```

```
Operations whose names satisfy the above pattern(s):

HermiteIntegrate       algintegrate           complexIntegrate
expintegrate           extendedIntegrate      fintegrate
infieldIntegrate       integrate              internalIntegrate
internalIntegrate0     lazyGintegrate         lazyIntegrate
lfintegrate            limitedIntegrate       monomialIntegrate
nagPolygonIntegrate    palgintegrate          pmComplexintegrate
pmintegrate            primintegrate          tanintegrate

To get more information about an operation such as
limitedIntegrate , issue the command )display op limitedIntegrate
```

1.15.1   Undo

A useful system command is)undo. Sometimes while computing interactively
with Axiom, you make a mistake and enter an incorrect definition or assignment.
Or perhaps you need to try one of several alternative approaches, one after
another, to find the best way to approach an application. For this, you will find
the *undo* facility of Axiom helpful.

System command)undo n means "undo back to step n"; it restores the values
of user variables to those that existed immediately after input expression n
was evaluated. Similarly,)undo -n undoes changes caused by the last n input
expressions. Once you have done an)undo, you can continue on from there, or
make a change and **redo** all your input expressions from the point of the)undo
forward. The)undo is completely general: it changes the environment like any
user expression. Thus you can)undo any previous undo.

Here is a sample dialogue between user and Axiom.

"Let me define two mutually dependent functions f and g piece-wise."

```
f(0) == 1; g(0) == 1
```

                                                           Type: Void

"Here is the general term for f."

```
f(n) == e/2*f(n-1) - x*g(n-1)
```

                                                           Type: Void

"And here is the general term for g."

```
g(n) == -x*f(n-1) + d/3*g(n-1)
```

                                                           Type: Void

"What is value of $f(3)$?"

```
f(3)
```

$$-x^3 + \left(e + \frac{1}{3}\ d\right) x^2 + \left(-\frac{1}{4}\ e^2 - \frac{1}{6}\ d\ e - \frac{1}{9}\ d^2\right) x + \frac{1}{8}\ e^3$$

                                  Type: Polynomial Fraction Integer

"Hmm, I think I want to define f differently. Undo to the environment right after I defined f."

```
)undo 2
```

"Here is how I think I want f to be defined instead."

```
f(n) == d/3*f(n-1) - x*g(n-1)
```

```
   1 old definition(s) deleted for function or rule f
```

```
                                                    Type:  Void
```

Redo the computation from expression 3 forward.

```
)undo )redo
```

```
g(n) == -x*f(n-1) + d/3*g(n-1)
```

```
                                                      Type: Void
```

```
f(3)
```

```
   Compiling function g with type Integer -> Polynomial Fraction
      Integer
   Compiling function g as a recurrence relation.
```

```
+++ |*1;g;1;G82322;AUX| redefined
```

```
+++ |*1;g;1;G82322| redefined
   Compiling function g with type Integer -> Polynomial Fraction
      Integer
   Compiling function g as a recurrence relation.
```

```
+++ |*1;g;1;G82322;AUX| redefined
```

```
+++ |*1;g;1;G82322| redefined
   Compiling function f with type Integer -> Polynomial Fraction
      Integer
   Compiling function f as a recurrence relation.
```

```
+++ |*1;f;1;G82322;AUX| redefined
```

```
+++ |*1;f;1;G82322| redefined
```

$$-x^3 + d\,x^2 - \frac{1}{3}\,d^2\,x + \frac{1}{27}\,d^3$$

<div align="right">Type: Polynomial Fraction Integer</div>

"I want my old definition of f after all. Undo the undo and restore the environment to that immediately after (4)."

```
)undo 4
```

"Check that the value of $f(3)$ is restored."

```
f(3)
```

```
   Compiling function g with type Integer -> Polynomial Fraction
      Integer
   Compiling function g as a recurrence relation.
```

```
+++ |*1;g;1;G82322;AUX| redefined
```

```
+++ |*1;g;1;G82322| redefined
   Compiling function g with type Integer -> Polynomial Fraction
      Integer
   Compiling function g as a recurrence relation.
```

```
+++ |*1;g;1;G82322;AUX| redefined
```

```
+++ |*1;g;1;G82322| redefined
   Compiling function f with type Integer -> Polynomial Fraction
      Integer
   Compiling function f as a recurrence relation.
```

```
+++ |*1;f;1;G82322;AUX| redefined
```

```
+++ |*1;f;1;G82322| redefined
```

$$-x^3 + \left(e + \frac{1}{3}\,d\right) x^2 + \left(-\frac{1}{4}\,e^2 - \frac{1}{6}\,d\,e - \frac{1}{9}\,d^2\right) x + \frac{1}{8}\,e^3$$

<div align="right">Type: Polynomial Fraction Integer</div>

After you have gone off on several tangents, then backtracked to previous points in your conversation using)undo, you might want to save all the "correct" input commands you issued, disregarding those undone. The system command)history)write mynew.input writes a clean straight-line program onto the file **mynew.input** on your disk.

1.16 Graphics

Axiom has a two- and three-dimensional drawing and rendering package that allows you to draw, shade, color, rotate, translate, map, clip, scale and combine graphic output of Axiom computations. The graphics interface is capable of plotting functions of one or more variables and plotting parametric surfaces. Once the graphics figure appears in a window, move your mouse to the window and click. A control panel appears immediately and allows you to interactively transform the object.

This is an example of Axiom's two-dimensional plotting. From the 2D Control Panel you can rescale the plot, turn axes and units on and off and save the image, among other things. This PostScript image was produced by clicking on the **PS** 2D Control Panel button.

```
draw(cos(5*t/8), t=0..16*%pi, coordinates==polar)
```
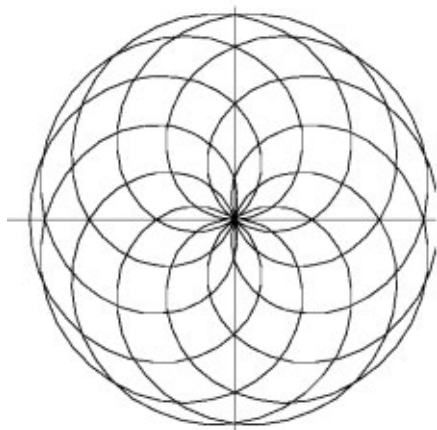

Figure 1.1: $J_0(\sqrt{x^2 + y^2})$ for $-20 \leq x, y \leq 20$

This is an example of Axiom's three-dimensional plotting. It is a monochrome graph of the complex arctangent function. The image displayed was rotated and had the "shade" and "outline" display options set from the 3D Control Panel. The PostScript output was produced by clicking on the **save** 3D Control Panel button and then clicking on the **PS** button. See Section for more details and examples of Axiom's numeric and graphics capabilities.

```
draw((x,y) +-> real atan complex(x,y), -%pi..%pi, -%pi..%pi,
colorFunction == (x,y) +-> argument atan complex(x,y))
```
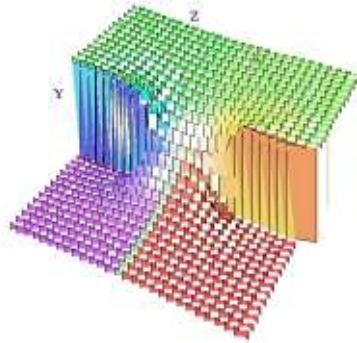
Figure 1.2: atan

An exhibit of Axiom images is given later. For a description of the commands and programs that produced these figures, see F on page 1077. PostScript output is available so that Axiom images can be printed.[14] See 7 on page 297 for more examples and details about using Axiom's graphics facilities.

This concludes your tour of Axiom. To disembark, issue the system command)quit to leave Axiom and return to the operating system.

---

[14]PostScript is a trademark of Adobe Systems Incorporated, registered in the United States.

Chapter 2

Using Types and Modes

Only recently have I begun to realize that the problem is not merely
one of technical mastery or the competent application of the rules
. . . but that there is actually something else which is guiding these
rules. It actually involves a different level of mastery. It's quite a
different process to do it right; and every single act that you do can
be done in that sense well or badly. But even assuming that you
have got the technical part clear, the creation of this quality is a
much more complicated process of the most utterly absorbing and
fascinating dimensions. It is in fact a major creative or artistic act
– every single little thing you do – . . .

– Christopher Alexander

(from Patterns of Software by Richard Gabriel)

In this chapter we look at the key notion of *type* and its generalization *mode*. We
show that every Axiom object has a type that determines what you can do with
the object. In particular, we explain how to use types to call specific functions
from particular parts of the library and how types and modes can be used to
create new objects from old. We also look at `Record` and `Union` types and the
special type `Any`. Finally, we give you an idea of how Axiom manipulates types
and modes internally to resolve ambiguities.

2.1   The Basic Idea

The Axiom world deals with many kinds of objects. There are mathematical
objects such as numbers and polynomials, data structure objects such as lists
and arrays, and graphics objects such as points and graphic images. Functions
are objects too.

Axiom organizes objects using the notion of domain of computation, or simply *domain*. Each domain denotes a class of objects. The class of objects it denotes is usually given by the name of the domain: `Integer` for the integers, `Float` for floating-point numbers, and so on. The convention is that the first letter of a domain name is capitalized. Similarly, the domain `Polynomial(Integer)` denotes "polynomials with integer coefficients." Also, `Matrix(Float)` denotes "matrices with floating-point entries."

Every basic Axiom object belongs to a unique domain. The integer 3 belongs to the domain `Integer` and the polynomial $x + 3$ belongs to the domain `Polynomial(Integer)`. The domain of an object is also called its *type*. Thus we speak of "the type `Integer`" and "the type `Polynomial(Integer)`."

After an Axiom computation, the type is displayed toward the right-hand side of the page (or screen).

`-3`

$$-3$$

Type: Integer

Here we create a rational number but it looks like the last result. The type however tells you it is different. You cannot identify the type of an object by how Axiom displays the object.

`-3/1`

$$-3$$

Type: Fraction Integer

When a computation produces a result of a simpler type, Axiom leaves the type unsimplified. Thus no information is lost.

`x + 3 - x`

$$3$$

Type: Polynomial Integer

This seldom matters since Axiom retracts the answer to the simpler type if it is necessary.

`factorial(%)`

6

<div align="right">Type: Expression Integer</div>

When you issue a positive number, the type `PositiveInteger` is printed. Surely, 3 also has type `Integer`! The curious reader may now have two questions. First, is the type of an object not unique? Second, how is `PositiveInteger` related to `Integer`?

3

3

<div align="right">Type: PositiveInteger</div>

Any domain can be refined to a *subdomain* by a membership `predicate`. A `predicate` is a function that, when applied to an object of the domain, returns either `true` or `false`. For example, the domain `Integer` can be refined to the subdomain `PositiveInteger`, the set of integers x such that $x > 0$, by giving the Axiom predicate $x + - > x > 0$. Similarly, Axiom can define subdomains such as "the subdomain of diagonal matrices," "the subdomain of lists of length two," "the subdomain of monic irreducible polynomials in x," and so on. Trivially, any domain is a subdomain of itself.

While an object belongs to a unique domain, it can belong to any number of subdomains. Any subdomain of the domain of an object can be used as the *type* of that object. The type of 3 is indeed both `Integer` and `PositiveInteger` as well as any other subdomain of integer whose predicate is satisfied, such as "the prime integers," "the odd positive integers between 3 and 17," and so on.

2.1.1 Domain Constructors

In Axiom, domains are objects. You can create them, pass them to functions, and, as we'll see later, test them for certain properties.

In Axiom, you ask for a value of a function by applying its name to a set of arguments.

To ask for "the factorial of 7" you enter this expression to Axiom. This applies the function `factorial` to the value 7 to compute the result.

`factorial(7)`

5040

<div align="right">Type: PositiveInteger</div>

Enter the type `Polynomial (Integer)` as an expression to Axiom. This looks
much like a function call as well. It is! The result is appropriately stated to be
of type `Domain`, which according to our usual convention, denotes the class of
all domains.

```
Polynomial(Integer)
```

                    Polynomial Integer

                                                    Type: Domain

The most basic operation involving domains is that of building a new domain
from a given one. To create the domain of "polynomials over the integers,"
Axiom applies the function `Polynomial` to the domain `Integer`. A function
like `Polynomial` is called a *domain constructor* or, more simply, a *constructor*.
A domain constructor is a function that creates a domain. An argument to a
domain constructor can be another domain or, in general, an arbitrary kind of
object. `Polynomial` takes a single domain argument while `SquareMatrix` takes
a positive integer as an argument to give its dimension and a domain argument
to give the type of its components.

What kinds of domains can you use as the argument to `Polynomial` or `SquareMatrix`
or `List`? Well, the first two are mathematical in nature. You want to be able
to perform algebraic operations like "`+`" and "`*`" on polynomials and square
matrices, and operations such as **determinant** on square matrices. So you
want to allow polynomials of integers *and* polynomials of square matrices with
complex number coefficients and, in general, anything that "makes sense." At
the same time, you don't want Axiom to be able to build nonsense domains
such as "polynomials of strings!"

In contrast to algebraic structures, data structures can hold any kind of object.
Operations on lists such as **insert**, **delete**, and **concat** just manipulate the list
itself without changing or operating on its elements. Thus you can build `List`
over almost any datatype, including itself.

Create a complicated algebraic domain.

```
List (List (Matrix (Polynomial (Complex (Fraction (Integer))))))
```

            List List Matrix Polynomial Complex Fraction Integer

                                                    Type: Domain

Try to create a meaningless domain.

```
Polynomial(String)
```

```
Polynomial String is not a valid type.
```

Evidently from our last example, Axiom has some mechanism that tells what a constructor can use as an argument. This brings us to the notion of *category*. As domains are objects, they too have a domain. The domain of a domain is a category. A category is simply a type whose members are domains.

A common algebraic category is `Ring`, the class of all domains that are "rings." A ring is an algebraic structure with constants 0 and 1 and operations "`+`", "`-`", and "`*`". These operations are assumed "closed" with respect to the domain, meaning that they take two objects of the domain and produce a result object also in the domain. The operations are understood to satisfy certain "axioms," certain mathematical principles providing the algebraic foundation for rings. For example, the *additive inverse axiom* for rings states:

> Every element x has an additive inverse y such that $x + y = 0$.

The prototypical example of a domain that is a ring is the integers. Keep them in mind whenever we mention `Ring`.

Many algebraic domain constructors such as `Complex`, `Polynomial`, `Fraction`, take rings as arguments and return rings as values. You can use the infix operator "*has*" to ask a domain if it belongs to a particular category.

All numerical types are rings. Domain constructor `Polynomial` builds "the ring of polynomials over any other ring."

```
Polynomial(Integer) has Ring
```

<div align="center">true</div>

<div align="right">Type: Boolean</div>

Constructor `List` never produces a ring.

```
List(Integer) has Ring
```

<div align="center">false</div>

<div align="right">Type: Boolean</div>

The constructor `Matrix(R)` builds "the domain of all matrices over the ring R." This domain is never a ring since the operations "`+`", "`-`", and "`*`" on matrices of arbitrary shapes are undefined.

```
Matrix(Integer) has Ring
```

false

Type: Boolean

Thus you can never build polynomials over matrices.

`Polynomial(Matrix(Integer))`

    Polynomial Matrix Integer is not a valid type.

Use `SquareMatrix(n,R)` instead. For any positive integer n, it builds "the ring of n by n matrices over R."

`Polynomial(SquareMatrix(7,Complex(Integer)))`

Polynomial SquareMatrix(7,Complex Integer)

Type: Domain

Another common category is `Field`, the class of all fields. A field is a ring with additional operations. For example, a field has commutative multiplication and a closed operation "/" for the division of two elements. `Integer` is not a field since, for example, 3/2 does not have an integer result. The prototypical example of a field is the rational numbers, that is, the domain `Fraction(Integer)`. In general, the constructor `Fraction` takes an IntegralDomain, which is a ring with additional properties, as an argument and returns a field. [1] Other domain constructors, such as `Complex`, build fields only if their argument domain is a field.

The complex integers (often called the "Gaussian integers") do not form a field.

`Complex(Integer) has Field`

false

Type: Boolean

But fractions of complex integers do.

`Fraction(Complex(Integer)) has Field`

true

---

[1] Actually, the argument domain must have some additional so as to belong to the category `IntegralDomain`

```
                                              Type:  Boolean
```

The algebraically equivalent domain of complex rational numbers is a field since domain constructor `Complex` produces a field whenever its argument is a field.

```
Complex(Fraction(Integer)) has Field
```

```
                        true
```

```
                                              Type:  Boolean
```

The most basic category is `Type`. It denotes the class of all domains and subdomains. Note carefully that `Type` does not denote the class of all types. The type of all categories is `Category`. The type of `Type` itself is undefined. Domain constructor `List` is able to build "lists of elements from domain D" for arbitrary D simply by requiring that D belong to category `Type`.

Now, you may ask, what exactly is a category? Like domains, categories can be defined in the Axiom language. A category is defined by three components:

1. a name (for example, `Ring`), used to refer to the class of domains that the category represents;

2. a set of operations, used to refer to the operations that the domains of this class support (for example, "`+`", "`-`", and "`*`" for rings); and

3. an optional list of other categories that this category extends.

This last component is a new idea. And it is key to the design of Axiom! Because categories can extend one another, they form hierarchies. Detailed charts showing the category hierarchies in Axiom are displayed in Appendix (TPDHERE). There you see that all categories are extensions of `Type` and that `Field` is an extension of `Ring`.

The operations supported by the domains of a category are called the *exports* of that category because these are the operations made available for system-wide use. The exports of a domain of a given category are not only the ones explicitly mentioned by the category. Since a category extends other categories, the operations of these other categories—and all categories these other categories extend—are also exported by the domains.

For example, polynomial domains belong to `PolynomialCategory`. This category explicitly mentions some twenty-nine operations on polynomials, but it extends eleven other categories (including `Ring`). As a result, the current system has over one hundred operations on polynomials.

If a domain belongs to a category that extends, say, `Ring`, it is convenient to say that the domain exports `Ring`. The name of the category thus provides a

convenient shorthand for the list of operations exported by the category. Rather than listing operations such as "`+`" and "`*`" of `Ring` each time they are needed, the definition of a type simply asserts that it exports category `Ring`.

The category name, however, is more than a shorthand. The name `Ring`, in fact, implies that the operations exported by rings are required to satisfy a set of "axioms" associated with the name `Ring`. This subtle but important feature distinguishes Axiom from other abstract datatype designs.

Why is it not correct to assume that some type is a ring if it exports all of the operations of `Ring`? Here is why. Some languages such as **APL** denote the `Boolean` constants `true` and `false` by the integers 1 and 0 respectively, then use "`+`" and "`*`" to denote the logical operators **or** and **and**. But with these definitions `Boolean` is not a ring since the additive inverse axiom is violated. That is, there is no inverse element a such that $1 + a = 0$, or, in the usual terms: `true or a = false`. This alternative definition of `Boolean` can be easily and correctly implemented in Axiom, since `Boolean` simply does not assert that it is of category `Ring`. This prevents the system from building meaningless domains such as `Polynomial(Boolean)` and then wrongfully applying algorithms that presume that the ring axioms hold.

Enough on categories. To learn more about them, see Chapter 12 on page 911. We now return to our discussion of domains.

Domains *export* a set of operations to make them available for system-wide use. `Integer`, for example, exports the operations "`+`" and "`=`" given by the signatures "`+`": (Integer,Integer) $\rightarrow$ Integer and "`=`": (Integer,Integer) $\rightarrow$ Boolean, respectively. Each of these operations takes two `Integer` arguments. The "`+`" operation also returns an `Integer` but "`=`" returns a `Boolean`: `true` or `false`. The operations exported by a domain usually manipulate objects of the domain—but not always.

The operations of a domain may actually take as arguments, and return as values, objects from any domain. For example, `Fraction (Integer)` exports the operations "`/`": (Integer,Integer) $\rightarrow$ Fraction(Integer) and **characteristic**: $\rightarrow$ NonNegativeInteger.

Suppose all operations of a domain take as arguments and return as values, only objects from *other* domains. This kind of domain is what Axiom calls a *package*.

A package does not designate a class of objects at all. Rather, a package is just a collection of operations. Actually the bulk of the Axiom library of algorithms consists of packages. The facilities for factorization; integration; solution of linear, polynomial, and differential equations; computation of limits; and so on, are all defined in packages. Domains needed by algorithms can be passed to a package as arguments or used by name if they are not "variable." Packages are useful for defining operations that convert objects of one type to another, particularly when these types have different parameterizations. As an example, the package `PolynomialFunction2(R,S)` defines operations that convert polynomials over a domain R to polynomials over S. To convert an object

from `Polynomial(Integer)` to `Polynomial(Float)`, Axiom builds the package `PolynomialFunctions2(Integer,Float)` in order to create the required conversion function. (This happens "behind the scenes" for you: see for details on how to convert objects.)

Axiom categories, domains and packages and all their contained functions are written in the Axiom programming language and have been compiled into machine code. This is what comprises the Axiom *library*. We will show you how to use these domains and their functions and how to write your own functions.

2.2 Writing Types and Modes

We have already seen in the last section several examples of types. Most of these examples had either no arguments (for example, `Integer`) or one argument (for example, `Polynomial (Integer)`). In this section we give details about writing arbitrary types. We then define modes and discuss how to write them. We conclude the section with a discussion on constructor abbreviations.

When might you need to write a type or mode? You need to do so when you declare variables.

```
a :  PositiveInteger
```

                                                              Type: Void

You need to do so when you declare functions (See Section),

```
f :  Integer -> String
```

                                                              Type: Void

You need to do so when you convert an object from one type to another (See Section).

```
factor(2 ::  Complex(Integer))
```

$$-i \ (1+i)^2$$

                                              Type: Factored Complex Integer

```
(2 = 3)$Integer
```

```
                                false
```

                                            Type: Boolean

You need to do so when you give computation target type information (See Section 2.9 on page 162).

```
(2 = 3)@Boolean
```

```
                                false
```

                                            Type: Boolean

2.2.1   Types with No Arguments

A constructor with no arguments can be written either with or without trailing opening and closing parentheses "()".

<div align="center">

`Boolean()` is the same as `Boolean`
`Integer()` is the same as `Integer`
`String()` is the same as `String`
`Void()` is the same as `Void`

</div>

It is customary to omit the parentheses.

2.2.2   Types with One Argument

A constructor with one argument can frequently be written with no parentheses. Types nest from right to left so that `Complex Fraction Polynomial Integer` is the same as `Complex (Fraction (Polynomial (Integer)))`. You need to use parentheses to force the application of a constructor to the correct argument, but you need not use any more than is necessary to remove ambiguities.

Here are some guidelines for using parentheses (they are possibly slightly more restrictive than they need to be).

If the argument is an expression like $2 + 3$ then you must enclose the argument in parentheses.

```
e :  PrimeField(2 + 3)
```

                                            Type: Void

If the type is to be used with package calling then you must enclose the argument in parentheses.

```
content(2)$Polynomial(Integer)
```

$$2$$

<div align="right">Type: Integer</div>

Alternatively, you can write the type without parentheses then enclose the whole type expression with parentheses.

```
content(2)$(Polynomial Complex Fraction Integer)
```

$$2$$

<div align="right">Type: Complex Fraction Integer</div>

If you supply computation target type information (See Section 2.9 on page 162) then you should enclose the argument in parentheses.

```
(2/3)@Fraction(Polynomial(Integer))
```

$$\frac{2}{3}$$

<div align="right">Type: Fraction Polynomial Integer</div>

If the type itself has parentheses around it and we are not in the case of the first example above, then the parentheses can usually be omitted.

```
(2/3)@Fraction(Polynomial Integer)
```

$$\frac{2}{3}$$

<div align="right">Type: Fraction Polynomial Integer</div>

If the type is used in a declaration and the argument is a single-word type, integer or symbol, then the parentheses can usually be omitted.

```
(d,f,g) :  Complex Polynomial Integer
```

<div align="right">Type: Void</div>

2.2.3   Types with More Than One Argument

If a constructor has more than one argument, you must use parentheses. Some
examples are

```
UnivariatePolynomial(x, Float)
MultivariatePolynomial([z,w,r], Complex Float)
SquareMatrix(3, Integer)
FactoredFunctions2(Integer,Fraction Integer)
```

2.2.4   Modes

A *mode* is a type that possibly is a question mark (`?`) or contains one in an
argument position. For example, the following are all modes.

```
?
Polynomial ?
Matrix Polynomial ?
SquareMatrix(3,?)
Integer
OneDimensionalArray(Float)
```

As is evident from these examples, a mode is a type with a part that is not
specified (indicated by a question mark). Only one "`?`" is allowed per mode
and it must appear in the most deeply nested argument that is a type. Thus
`?(Integer)`, `Matrix(?  (Polynomial))`, `SquareMatrix(?, Integer)` (it re-
quires a numeric argument) and `SquareMatrix(?, ?)` are all invalid. The
question mark must take the place of a domain, not data. This rules out, for
example, the two `SquareMatrix` expressions.

Modes can be used for declarations (See Section 2.3 on page 142) and conversions
(Section 2.7 on page 155). However, you cannot use a mode for package calling
or giving target type information.

2.2.5   Abbreviations

Every constructor has an abbreviation that you can freely substitute for the
constructor name. In some cases, the abbreviation is nothing more than the
capitalized version of the constructor name.

Aside from allowing types to be written more concisely, abbreviations are used by Axiom to name various system files for constructors (such as library filenames, test input files and example files). Here are some common abbreviations.

`COMPLEX` abbreviates `Complex`	`DFLOAT` abbreviates `DoubleFloat`
`EXPR` abbreviates `Expression`	`FLOAT` abbreviates `Float`
`FRAC` abbreviates `Fraction`	`INT` abbreviates `Integer`
`MATRIX` abbreviates `Matrix`	`NNI` abbreviates `NonNegativeInteger`
`PI` abbreviates `PositiveInteger`	`POLY` abbreviates `Polynomial`
`STRING` abbreviates `String`	`UP` abbreviates `UnivariatePolynomial`

You can combine both full constructor names and abbreviations in a type expression. Here are some types using abbreviations.

`POLY INT`	is the same as	`Polynomial(INT)`
`POLY(Integer)`	is the same as	`Polynomial(Integer)`
`POLY(Integer)`	is the same as	`Polynomial(INT)`
`FRAC(COMPLEX(INT))`	is the same as	`Fraction Complex Integer`
`FRAC(COMPLEX(INT))`	is the same as	`FRAC(Complex Integer)`

There are several ways of finding the names of constructors and their abbreviations. For a specific constructor, use `)abbreviation query`. You can also use the `)what` system command to see the names and abbreviations of constructors. For more information about `)what`, see **??** on page **??**.

`)abbreviation query` can be abbreviated (no pun intended) to `)abb q`.

```
)abb q Integer
```

```
   INT abbreviates domain Integer
```

The `)abbreviation query` command lists the constructor name if you give the abbreviation. Issue `)abb q` if you want to see the names and abbreviations of all Axiom constructors.

```
)abb q DMP
```

```
   DMP abbreviates domain DistributedMultivariatePolynomial
```

Issue this to see all packages whose names contain the string "ode".

```
)what packages ode
```

```
---------------------- Packages -----------------------

Packages with names matching patterns:
     ode

 EXPRODE  ExpressionSpaceODESolver
 FCPAK1   FortranCodePackage1
 GRAY     GrayCode
 LODEEF   ElementaryFunctionLODESolver
 NODE1    NonLinearFirstOrderODESolver
 ODECONST ConstantLODE
 ODEEF    ElementaryFunctionODESolver
 ODEINT   ODEIntegration
 ODEPAL   PureAlgebraicLODE
 ODERAT   RationalLODE
 ODERED   ReduceLODE
 ODESYS   SystemODESolver
 ODETOOLS ODETools
 UTSODE   UnivariateTaylorSeriesODESolver
 UTSODETL UTSodetools
```

2.3   Declarations

A *declaration* is an expression used to restrict the type of values that can be
assigned to variables. A colon ":" is always used after a variable or list of
variables to be declared.

For a single variable, the syntax for declaration is

$$variableName : typeOrMode$$

For multiple variables, the syntax is

$$(variableName_1,\ variableName_2,\ \ldots variableName_N):\quad typeOrMode$$

You can always combine a declaration with an assignment. When you do, it is
equivalent to first giving a declaration statement, then giving an assignment.
For more information on assignment, see Section 1.3.4 on page 77 and Section
5.1 on page 195. To see how to declare your own functions, see 6.4 on page 237.

This declares one variable to have a type.

```
a :  Integer
```

                                                              Type: Void

This declares several variables to have a type.

```
(b,c) :  Integer
```

<div align="right">Type: Void</div>

a, b and c can only hold integer values.

```
a := 45
```

<div align="center">45</div>

<div align="right">Type: Integer</div>

If a value cannot be converted to a declared type, an error message is displayed.

```
b := 4/5
```

```
   Cannot convert right-hand side of assignment
   4
   -
   5

     to an object of the type Integer of the left-hand side.
```

This declares a variable with a mode.

```
n :  Complex ?
```

<div align="right">Type: Void</div>

This declares several variables with a mode.

```
(p,q,r) :  Matrix Polynomial ?
```

<div align="right">Type: Void</div>

This complex object has integer real and imaginary parts.

```
n := -36 + 9 * %i
```

$$-36 + 9\,i$$

<div align="right">Type: Complex Integer</div>

This complex object has fractional symbolic real and imaginary parts.

```
n := complex(4/(x + y),y/x)
```

$$\frac{4}{y + x} + \frac{y}{x}\,i$$

<div align="right">Type: Complex Fraction Polynomial Integer</div>

This matrix has entries that are polynomials with integer coefficients.

```
p := [ [1,2],[3,4],[5,6] ]
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

<div align="right">Type: Matrix Polynomial Integer</div>

This matrix has a single entry that is a polynomial with rational number coefficients.

```
q := [ [x - 2/3] ]
```

$$\begin{bmatrix} x - \frac{2}{3} \end{bmatrix}$$

<div align="right">Type: Matrix Polynomial Fraction Integer</div>

This matrix has entries that are polynomials with complex integer coefficients.

```
r := [ [1-%i*x,7*y+4*%i] ]
```

$$\begin{bmatrix} -i\,x + 1 & 7\,y + 4\,i \end{bmatrix}$$

<div align="right">Type: Matrix Polynomial Complex Integer</div>

Note the difference between this and the next example. This is a complex object with polynomial real and imaginary parts.

```
f :  COMPLEX POLY ? := (x + y*%i)**2
```

$$-y^2 + x^2 + 2\ x\ y\ i$$

<div align="right">

Type: Complex Polynomial Integer

</div>

This is a polynomial with complex integer coefficients. The objects are convertible from one to the other. See 2.7 on page 155 for more information.

```
g :  POLY COMPLEX ? := (x + y*%i)**2
```

$$-y^2 + 2\ i\ x\ y + x^2$$

<div align="right">

Type: Polynomial Complex Integer

</div>

2.4   Records

A `Record` is an object composed of one or more other objects, each of which is referenced with a *selector*. Components can all belong to the same type or each can have a different type.

The syntax for writing a `Record` type is

   `Record(`*selector$_1$* `:` *type$_1$*, *selector$_2$* `:` *type$_2$*, `...`, *selector$_N$* `:` *type$_N$*`)`

You must be careful if a selector has the same name as a variable in the workspace. If this occurs, precede the selector name by a single quote.

Record components are implicitly ordered. All the components of a record can be set at once by assigning the record a bracketed *tuple* of values of the proper length. For example:

```
r :  Record(a:Integer, b:  String) := [1, "two"]
```

$$[a = 1, b = \texttt{"two"}]$$

<div align="right">

Type: Record(a: Integer,b: String)

</div>

To access a component of a record r, write the name r, followed by a period, followed by a selector.

The object returned by this computation is a record with two components: a *quotient* part and a *remainder* part.

```
u := divide(5,2)
```

$$[quotient = 2, remainder = 1]$$

                    Type: Record(quotient: Integer,remainder: Integer)

This is the quotient part.

u.quotient

$$2$$

                                    Type: PositiveInteger

This is the remainder part.

u.remainder

$$1$$

                                    Type: PositiveInteger

You can use selector expressions on the left-hand side of an assignment to change destructively the components of a record.

u.quotient := 8978

$$8978$$

                                    Type: PositiveInteger

The selected component *quotient* has the value 8978, which is what is returned by the assignment. Check that the value of *u* was modified.

u

$$[quotient = 8978, remainder = 1]$$

                    Type: Record(quotient: Integer,remainder: Integer)

Selectors are evaluated. Thus you can use variables that evaluate to selectors instead of the selectors themselves.

s := 'quotient

$$quotient$$

```
                                              Type:  Variable quotient
```

Be careful! A selector could have the same name as a variable in the workspace. If this occurs, precede the selector name by a single quote, as in *u.'quotient*.

```
divide(5,2).s
```

$$2$$

```
                                              Type:  PositiveInteger
```

Here we declare that the value of *bd* has two components: a string, to be accessed via `name`, and an integer, to be accessed via `birthdayMonth`.

```
bd :  Record(name :  String, birthdayMonth :  Integer)
```

```
                                                            Type:  Void
```

You must initially set the value of the entire `Record` at once.

```
bd := ["Judith", 3]
```

$$[name = \texttt{"Judith"}, birthdayMonth = 3]$$

```
          Type:  Record(name:  String,birthdayMonth:  Integer)
```

Once set, you can change any of the individual components.

```
bd.name := "Katie"
```

```
                             "Katie"
```

```
                                                          Type:  String
```

Records may be nested and the selector names can be shared at different levels.

```
r :  Record(a :  Record(b:  Integer, c:  Integer), b:  Integer)
```

```
                                                            Type:  Void
```

The record *r* has a *b* selector at two different levels. Here is an initial value for *r*.

```
r := [ [1,2], 3 ]
```

$$[a = [b = 1, c = 2], b = 3]$$

Type: Record(a: Record(b: Integer,c: Integer),b: Integer)

This extracts the b component from the a component of r.

```
r.a.b
```

1

Type: PositiveInteger

This extracts the b component from r.

```
r.b
```

3

Type: PositiveInteger

You can also use spaces or parentheses to refer to `Record` components. This is the same as $r.a$.

```
r(a)
```

$$[b = 1, c = 2]$$

Type: Record(b: Integer,c: Integer)

This is the same as $r.b$.

```
r b
```

3

Type: PositiveInteger

This is the same as $r.b := 10$.

```
r(b) := 10
```

10

```
                                      Type:  PositiveInteger
```

Look at r to make sure it was modified.

```
r
```

$$[a = [b = 1, c = 2], b = 10]$$

```
   Type:  Record(a:  Record(b:  Integer,c:  Integer),b:  Integer)
```

2.5   Unions

Type `Union` is used for objects that can be of any of a specific finite set of types.
Two versions of unions are available, one with selectors (like records) and one
without.

2.5.1   Unions Without Selectors

The declaration $x : Union(Integer, String, Float)$ states that x can have values
that are integers, strings or "big" floats. If, for example, the `Union` object is an
integer, the object is said to belong to the `Integer` *branch* of the `Union`. Note
that we are being a bit careless with the language here. Technically, the type
of x is always `Union(Integer, String, Float)`. If it belongs to the `Integer`
branch, x may be converted to an object of type `Integer`.

The syntax for writing a `Union` type without selectors is

$$\texttt{Union}(type_1, \ type_2, \ \ldots, \ type + N)$$

The types in a union without selectors must be distinct.

It is possible to create unions like `Union(Integer, PositiveInteger)` but they
are difficult to work with because of the overlap in the branch types. See below
for the rules Axiom uses for converting something into a union object.

The `case` infix operator returns a `Boolean` and can be used to determine the
branch in which an object lies.

This function displays a message stating in which branch of the `Union` the object
(defined as x above) lies.

```
sayBranch(x : Union(Integer,String,Float)) : Void  ==
  output
```

```
    x case Integer => "Integer branch"
    x case String  => "String branch"
    "Float branch"
```

This tries **sayBranch** with an integer.

```
sayBranch 1
```

```
Compiling function sayBranch with type Union(Integer,String,Float)
    -> Void
 Integer branch
```

                                                          Type: Void

This tries **sayBranch** with a string.

```
sayBranch "hello"
```

```
   String branch
```

                                                          Type: Void

This tries **sayBranch** with a floating-point number.

```
sayBranch 2.718281828
```

```
   Float branch
```

                                                          Type: Void

There are two things of interest about this particular example to which we would
like to draw your attention.

1. Axiom normally converts a result to the target value before passing it to
    the function. If we left the declaration information out of this function
    definition then the **sayBranch** call would have been attempted with an
    `Integer` rather than a `Union`, and an error would have resulted.

2. The types in a `Union` are searched in the order given. So if the type were
    given as

    ```
    sayBranch(x:  Union(String,Integer,Float,Any)):  Void
    ```

    then the result would have been "String branch" because there is a con-
    version from `Integer` to `String`.

Sometimes `Union` types can have extremely long names. Axiom therefore abbreviates the names of unions by printing the type of the branch first within the `Union` and then eliding the remaining types with an ellipsis (. . .).

Here the `Integer` branch is displayed first. Use "`::`" to create a `Union` object from an object.

```
78 ::  Union(Integer,String)
```

$$78$$

                                    Type: Union(Integer,...)

Here the `String` branch is displayed first.

```
s := "string" ::  Union(Integer,String)
```

$$\text{"string"}$$

                                    Type: Union(String,...)

Use `typeOf` to see the full and actual `Union` type.

```
typeOf s
```

$$Union(Integer, String)$$

                                         Type: Domain

A common operation that returns a union is **exquo** which returns the "exact quotient" if the quotient is exact,

```
three := exquo(6,2)
```

$$3$$

                                    Type: Union(Integer,...)

and `"failed"` if the quotient is not exact.

```
exquo(5,2)
```

$$\text{"failed"}$$

                                    Type: Union("failed",...)

A union with a `"failed"` is frequently used to indicate the failure or lack of
applicability of an object. As another example, assign an integer a variable r
declared to be a rational number.

```
r:  FRAC INT := 3
```

$$3$$

<div align="right">Type: Fraction Integer</div>

The operation **retractIfCan** tries to retract the fraction to the underlying
domain `Integer`. It produces a union object. Here it succeeds.

```
retractIfCan(r)
```

$$3$$

<div align="right">Type: Union(Integer,...)</div>

Assign it a rational number.

```
r := 3/2
```

$$\frac{3}{2}$$

<div align="right">Type: Fraction Integer</div>

Here the retraction fails.

```
retractIfCan(r)
```

$$\text{"failed"}$$

<div align="right">Type: Union("failed",...)</div>

2.5.2   Unions With Selectors

Like records (2.4 on page 145), you can write `Union` types with selectors.

The syntax for writing a `Union` type with selectors is

$$\texttt{Union}(selector_1\!:\!type_1,\ selector_2\!:\!type_2,\ \ldots,\ selector_N\!:\!type_N)$$

You must be careful if a selector has the same name as a variable in the
workspace. If this occurs, precede the selector name by a single quote. It
is an error to use a selector that does not correspond to the branch of the
`Union` in which the element actually lies.

Be sure to understand the difference between records and unions with selectors. Records can have more than one component and the selectors are used to refer to the components. Unions always have one component but the type of that one component can vary. An object of type `Record(a:  Integer, b:  Float, c:  String)` contains an integer *and* a float *and* a string. An object of type `Union(a:  Integer, b:  Float, c:  String)` contains an integer *or* a float *or* a string.

Here is a version of the **sayBranch** function (cf. 2.5.1 on page 149) that works with a union with selectors. It displays a message stating in which branch of the `Union` the object lies.

```
sayBranch(x:Union(i:Integer,s:String,f:Float)):Void==
  output
    x case i => "Integer branch"
    x case s  => "String branch"
    "Float branch"
```

Note that `case` uses the selector name as its right-hand argument. If you accidentally use the branch type on the right-hand side of `case`, `false` will be returned.

Declare variable *u* to have a union type with selectors.

```
u :  Union(i :  Integer, s :  String)
```

$$\text{Type: Void}$$

Give an initial value to *u*.

```
u := "good morning"
```

$$\text{"good morning"}$$

$$\text{Type: Union(s: String,...)}$$

Use *case* to determine in which branch of a `Union` an object lies.

```
u case i
```

$$\text{false}$$

$$\text{Type: Boolean}$$

```
u case s
```

                                    true

                                                    Type: Boolean

To access the element in a particular branch, use the selector.

u.s

                              "good morning"

                                                    Type: String

2.6   The "Any" Domain

With the exception of objects of type `Record`, all Axiom data structures are
homogenous, that is, they hold objects all of the same type. If you need to get
around this, you can use type `Any`. Using `Any`, for example, you can create lists
whose elements are integers, rational numbers, strings, and even other lists.

Declare u to have type `Any`.

u: Any

                                                    Type: Void

Assign a list of mixed type values to u

u := [1, 7.2, 3/2, x**2, "wally"]

$$\left[1, 7.2, \frac{3}{2}, x^2, \texttt{"wally"}\right]$$

                                                    Type: List Any

When we ask for the elements, Axiom displays these types.

u.1

                                     1

                                                    Type: PositiveInteger

Actually, these objects belong to `Any` but Axiom automatically converts them to their natural types for you.

u.3

$$\frac{3}{2}$$

Type: Fraction Integer

Since type `Any` can be anything, it can only belong to type `Type`. Therefore it cannot be used in algebraic domains.

v : Matrix(Any)

    Matrix Any is not a valid type.

Perhaps you are wondering how Axiom internally represents objects of type `Any`. An object of type `Any` consists not only a data part representing its normal value, but also a type part (a *badge*) giving its type. For example, the value 1 of type `PositiveInteger` as an object of type `Any` internally looks like $[1, \texttt{PositiveInteger}()]$.

When should you use `Any` instead of a `Union` type? For a `Union`, you must know in advance exactly which types you are going to allow. For `Any`, anything that comes along can be accommodated.

2.7 Conversion

Conversion is the process of changing an object of one type into an object of another type. The syntax for conversion is:

$$object::newType$$

By default, 3 has the type `PositiveInteger`.

3

$$3$$

Type: PositiveInteger

We can change this into an object of type `Fraction Integer` by using "::".

```
3 ::  Fraction Integer
```

$$3$$

<div align="right">Type: Fraction Integer</div>

A *coercion* is a special kind of conversion that Axiom is allowed to do automatically when you enter an expression. Coercions are usually somewhat safer than more general conversions. The Axiom library contains operations called **coerce** and **convert**. Only the **coerce** operations can be used by the interpreter to change an object into an object of another type unless you explicitly use a `::`.

By now you will be quite familiar with what types and modes look like. It is useful to think of a type or mode as a pattern for what you want the result to be.

Let's start with a square matrix of polynomials with complex rational number coefficients.

```
m :  SquareMatrix(2,POLY COMPLEX FRAC INT)
```

<div align="right">Type: Void</div>

```
m := matrix [ [x-3/4*%i,z*y**2+1/2],[3/7*%i*y**4 - x,12-%i*9/5] ]
```

$$\begin{bmatrix} x - \frac{3}{4}\ i & y^2\ z + \frac{1}{2} \\ \frac{3}{7}\ i\ y^4 - x & 12 - \frac{9}{5}\ i \end{bmatrix}$$

<div align="center">Type: SquareMatrix(2,Polynomial Complex Fraction Integer)</div>

We first want to interchange the `Complex` and `Fraction` layers. We do the conversion by doing the interchange in the type expression.

```
m1 := m ::  SquareMatrix(2,POLY FRAC COMPLEX INT)
```

$$\begin{bmatrix} x - \frac{3\ i}{4} & y^2\ z + \frac{1}{2} \\ \frac{3\ i}{7}\ y^4 - x & \frac{60 - 9\ i}{5} \end{bmatrix}$$

<div align="center">Type: SquareMatrix(2,Polynomial Fraction Complex Integer)</div>

Interchange the `Polynomial` and the `Fraction` levels.

```
m2 := m1 ::  SquareMatrix(2,FRAC POLY COMPLEX INT)
```

$$\left[\begin{array}{cc} \frac{4\ x-3\ i}{4} & \frac{2\ y^2\ z+1}{2} \\ \frac{3\ i\ y^4-7\ x}{7} & \frac{60-9\ i}{5} \end{array} \right]$$

Type: SquareMatrix(2,Fraction Polynomial Complex Integer)

Interchange the `Polynomial` and the `Complex` levels.

m3 := m2 :: SquareMatrix(2,FRAC COMPLEX POLY INT)

$$\left[\begin{array}{cc} \frac{4\ x-3\ i}{4} & \frac{2\ y^2\ z+1}{2} \\ \frac{-7\ x+3\ y^4\ i}{7} & \frac{60-9\ i}{5} \end{array} \right]$$

Type: SquareMatrix(2,Fraction Complex Polynomial Integer)

All the entries have changed types, although in comparing the last two results only the entry in the lower left corner looks different. We did all the intermediate steps to show you what Axiom can do.

In fact, we could have combined all these into one conversion.

m :: SquareMatrix(2,FRAC COMPLEX POLY INT)

$$\left[\begin{array}{cc} \frac{4\ x-3\ i}{4} & \frac{2\ y^2\ z+1}{2} \\ \frac{-7\ x+3\ y^4\ i}{7} & \frac{60-9\ i}{5} \end{array} \right]$$

Type: SquareMatrix(2,Fraction Complex Polynomial Integer)

There are times when Axiom is not be able to do the conversion in one step. You may need to break up the transformation into several conversions in order to get an object of the desired type.

We cannot move either `Fraction` or `Complex` above (or to the left of, depending on how you look at it) `SquareMatrix` because each of these levels requires that its argument type have commutative multiplication, whereas `SquareMatrix` does not. That is because `Fraction` requires that its argument belong to the category `IntegralDomain` and `Complex` requires that its argument belong to `CommutativeRing`. See 2.1 on page 129 for a brief discussion of categories. The `Integer` level did not move anywhere because it does not allow any arguments. We also did not move the `SquareMatrix` part anywhere, but we could have.

Recall that m looks like this.

m

$$\left[\begin{array}{cc} x-\frac{3}{4}\ i & y^2\ z+\frac{1}{2} \\ \frac{3}{7}\ i\ y^4-x & 12-\frac{9}{5}\ i \end{array} \right]$$

                    Type: SquareMatrix(2,Polynomial Complex Fraction Integer)

If we want a polynomial with matrix coefficients rather than a matrix with polynomial entries, we can just do the conversion.

```
m ::  POLY SquareMatrix(2,COMPLEX FRAC INT)
```

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} y^2\, z + \begin{bmatrix} 0 & 0 \\ \frac{3}{7}\,i & 0 \end{bmatrix} y^4 + \begin{bmatrix} 1 & 0 \\ -1 & 0 \end{bmatrix} x + \begin{bmatrix} -\frac{3}{4}\,i & \frac{1}{2} \\ 0 & 12 - \frac{9}{5}\,i \end{bmatrix}$$

              Type: Polynomial SquareMatrix(2,Complex Fraction Integer)

We have not yet used modes for any conversions. Modes are a great shorthand for indicating the type of the object you want. Instead of using the long type expression in the last example, we could have simply said this.

```
m ::  POLY ?
```

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} y^2\, z + \begin{bmatrix} 0 & 0 \\ \frac{3}{7}\,i & 0 \end{bmatrix} y^4 + \begin{bmatrix} 1 & 0 \\ -1 & 0 \end{bmatrix} x + \begin{bmatrix} -\frac{3}{4}\,i & \frac{1}{2} \\ 0 & 12 - \frac{9}{5}\,i \end{bmatrix}$$

              Type: Polynomial SquareMatrix(2,Complex Fraction Integer)

We can also indicate more structure if we want the entries of the matrices to be fractions.

```
m ::  POLY SquareMatrix(2,FRAC ?)
```

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} y^2\, z + \begin{bmatrix} 0 & 0 \\ \frac{3\,i}{7} & 0 \end{bmatrix} y^4 + \begin{bmatrix} 1 & 0 \\ -1 & 0 \end{bmatrix} x + \begin{bmatrix} -\frac{3\,i}{4} & \frac{1}{2} \\ 0 & \frac{60 - 9\,i}{5} \end{bmatrix}$$

              Type: Polynomial SquareMatrix(2,Fraction Complex Integer)

2.8   Subdomains Again

A *subdomain* S of a domain D is a domain consisting of

1. those elements of D that satisfy some *predicate* (that is, a test that returns true or false) and

2. a subset of the operations of D.

Every domain is a subdomain of itself, trivially satisfying the membership test: `true`.

Currently, there are only two system-defined subdomains in Axiom that receive substantial use. `PositiveInteger` and `NonNegativeInteger` are subdomains of `Integer`. An element x of `NonNegativeInteger` is an integer that is greater than or equal to zero, that is, satisfies $x >= 0$. An element x of `PositiveInteger` is a nonnegative integer that is, in fact, greater than zero, that is, satisfies $x > 0$. Not all operations from `Integer` are available for these subdomains. For example, negation and subtraction are not provided since the subdomains are not closed under those operations. When you use an integer in an expression, Axiom assigns to it the type that is the most specific subdomain whose predicate is satisfied.

This is a positive integer.

```
5
```

$$5$$

                                        Type: PositiveInteger

This is a nonnegative integer.

```
0
```

$$0$$

                                        Type: NonNegativeInteger

This is neither of the above.

```
-5
```

$$-5$$

                                        Type: Integer

Furthermore, unless you are assigning an integer to a declared variable or using a conversion, any integer result has as type the most specific subdomain.

```
(-2) - (-3)
```

$$1$$

                                        Type: PositiveInteger

```
0 ::  Integer
```

$$0$$

<div align="right">Type: Integer</div>

```
x :  NonNegativeInteger := 5
```

$$5$$

<div align="right">Type: NonNegativeInteger</div>

When necessary, Axiom converts an integer object into one belonging to a less specific subdomain. For example, in $3 - 2$, the arguments to "-" are both elements of `PositiveInteger`, but this type does not provide a subtraction operation. Neither does `NonNegativeInteger`, so 3 and 2 are viewed as elements of `Integer`, where their difference can be calculated. The result is 1, which Axiom then automatically assigns the type `PositiveInteger`.

Certain operations are very sensitive to the subdomains to which their arguments belong. This is an element of `PositiveInteger`.

```
2 ** 2
```

$$4$$

<div align="right">Type: PositiveInteger</div>

This is an element of `Fraction Integer`.

```
2 ** (-2)
```

$$\frac{1}{4}$$

<div align="right">Type: Fraction Integer</div>

It makes sense then that this is a list of elements of `PositiveInteger`.

```
[10**i for i in 2..5]
```

$$[100, 1000, 10000, 100000]$$

<div align="right">Type: List PositiveInteger</div>

What should the type of `[10**(i-1) for i in 2..5]` be? On one hand, $i-1$ is always an integer greater than zero as i ranges from 2 to 5 and so $10**i$ is also always a positive integer. On the other, $i-1$ is a very simple function of i. Axiom does not try to analyze every such function over the index's range of values to determine whether it is always positive or nowhere negative. For an arbitrary Axiom function, this analysis is not possible.

So, to be consistent no such analysis is done and we get this.

```
[10**(i-1) for i in 2..5]
```

$$[10, 100, 1000, 10000]$$

```
                              Type:  List Fraction Integer
```

To get a list of elements of `PositiveInteger` instead, you have two choices. You can use a conversion.

```
[10**((i-1) ::  PI) for i in 2..5]
```

```
Compiling function G82696 with type Integer -> Boolean
Compiling function G82708 with type NonNegativeInteger -> Boolean
```

$$[10, 100, 1000, 10000]$$

```
                                Type:  List PositiveInteger
```

Or you can use `pretend`.

```
[10**((i-1) pretend PI) for i in 2..5]
```

$$[10, 100, 1000, 10000]$$

```
                                Type:  List PositiveInteger
```

The operation `pretend` is used to defeat the Axiom type system. The expression `object pretend D` means "make a new object (without copying) of type `D` from `object`." If `object` were an integer and you told Axiom to pretend it was a list, you would probably see a message about a fatal error being caught and memory possibly being damaged. Lists do not have the same internal representation as integers!

You use `pretend` at your peril.

Use *pretend* with great care! Axiom trusts you that the value is of the specified type.

`(2/3) pretend Complex Integer`

$$2 + 3\ i$$

Type: Complex Integer

2.9   Package Calling and Target Types

Axiom works hard to figure out what you mean by an expression without your having to qualify it with type information. Nevertheless, there are times when you need to help it along by providing hints (or even orders!) to get Axiom to do what you want.

We saw in 2.3 on page 142 that declarations using types and modes control the type of the results produced. For example, we can either produce a complex object with polynomial real and imaginary parts or a polynomial with complex integer coefficients, depending on the declaration.

Package calling is how you tell Axiom to use a particular function from a particular part of the library.

Use the "/" from `Fraction Integer` to create a fraction of two integers.

`2/3`

$$\frac{2}{3}$$

Type: Fraction Integer

If we wanted a floating point number, we can say "use the "/" in `Float`."

`(2/3)$Float`

$$0.66666666666666666667$$

Type: Float

Perhaps we actually wanted a fraction of complex integers.

`(2/3)$Fraction(Complex Integer)`

$$\frac{2}{3}$$

Type: Float

In each case, AXIOM used the indicated operations, sometimes first needing to convert the two integers into objects of the appropriate type. In these examples, "/" is written as an infix operator.

To use package calling with an infix operator, use the following syntax:

$$(\ arg_1 \ \text{op} \ arg_2 \ )\$type$$

We used, for example, $(2/3)\$$Float. The expression $2 + 3 + 4$ is equivalent to $(2+3)+4$. Therefore in the expression $(2+3+4)\$$Float the second "+" comes from the Float domain. The first "+" comes from Float because the package call causes AXIOM to convert $(2 + 3)$ and 4 to type Float. Before the sum is converted, it is given a target type of Float by AXIOM and then evaluated. The target type causes the "+" from `Float` to be used.

For an operator written before its arguments, you must use parentheses around the arguments (even if there is only one), and follow the closing parenthesis by a "$" and then the type.

$$fun \ (\ arg_1, arg_2, \ldots, arg_N \ )\$type$$

For example, to call the "minimum" function from SmallFloat on two integers, you could write **min**(4,89)`SmallFloat`. Another use of package calling is to tell AXIOM to use a library function rather than a function you defined. We discuss this in Section 6.9 on page 244.

Sometimes rather than specifying where an operation comes from, you just want to say what type the result should be. We say that you provide a *target type* for the expression. Instead of using a "$", use a "@" to specify the requested target type. Otherwise, the syntax is the same. Note that giving a target type is not the same as explicitly doing a conversion. The first says "try to pick operations so that the result has such-and-such a type." The second says "compute the result and then convert to an object of such-and-such a type."

Sometimes it makes sense, as in this expression, to say "choose the operations in this expression so that the final result is Float.

```
(2/3)@Float
```

$$0.66666666666666666667$$

Type: Float

Here we used "@" to say that the target type of the left-hand side was `Float`. In this simple case, there was no real difference between using "$" and "@". You can see the difference if you try the following.

This says to try to choose "+" so that the result is a string. Axiom cannot do this.

```
(2 + 3)@String
```

```
An expression involving @ String actually evaluated to one of
    type PositiveInteger . Perhaps you should use :: String .
```

This says to get the + from `String` and apply it to the two integers. Axiom also cannot do this because there is no + exported by `String`.

```
(2 + 3)$String
```

```
    The function + is not implemented in String .
```

(By the way, the operation **concat** or juxtaposition is used to concatenate two strings.)

When we have more than one operation in an expression, the difference is even more evident. The following two expressions show that Axiom uses the target type to create different objects. The "+", "*" and "**" operations are all chosen so that an object of the correct final type is created.

This says that the operations should be chosen so that the result is a `Complex` object.

```
((x + y * %i)**2)@(Complex Polynomial Integer)
```

$$-y^2 + x^2 + 2\ x\ y\ i$$

<div align="right">Type: Complex Polynomial Integer</div>

This says that the operations should be chosen so that the result is a `Polynomial` object.

```
((x + y * %i)**2)@(Polynomial Complex Integer)
```

$$-y^2 + 2\ i\ x\ y + x^2$$

<div align="right">Type: Polynomial Complex Integer</div>

What do you think might happen if we left off all target type and package call information in this last example?

```
(x + y * %i)**2
```

$$-y^2 + 2\ i\ x\ y + x^2$$

<div align="right">Type: Polynomial Complex Integer</div>

We can convert it to `Complex` as an afterthought. But this is more work than just saying making what we want in the first place.

```
% ::  Complex ?
```

$$-y^2 + x^2 + 2\ x\ y\ i$$

<div align="right">Type: Complex Polynomial Integer</div>

Finally, another use of package calling is to qualify fully an operation that is passed as an argument to a function.

Start with a small matrix of integers.

```
h := matrix [ [8,6],[-4,9] ]
```

$$\begin{bmatrix} 8 & 6 \\ -4 & 9 \end{bmatrix}$$

<div align="right">Type: Matrix Integer</div>

We want to produce a new matrix that has for entries the multiplicative inverses of the entries of h. One way to do this is by calling **map** with the **inv** function from `Fraction (Integer)`.

```
map(inv$Fraction(Integer),h)
```

$$\begin{bmatrix} \frac{1}{8} & \frac{1}{6} \\ -\frac{1}{4} & \frac{1}{9} \end{bmatrix}$$

<div align="right">Type: Matrix Fraction Integer</div>

We could have been a bit less verbose and used abbreviations.

```
map(inv$FRAC(INT),h)
```

$$\begin{bmatrix} \frac{1}{8} & \frac{1}{6} \\ -\frac{1}{4} & \frac{1}{9} \end{bmatrix}$$

<div align="right">Type: Matrix Fraction Integer</div>

As it turns out, Axiom is smart enough to know what we mean anyway. We can just say this.

```
map(inv,h)
```

$$\begin{bmatrix} \frac{1}{8} & \frac{1}{6} \\ -\frac{1}{4} & \frac{1}{9} \end{bmatrix}$$

<div align="right">Type: Matrix Fraction Integer</div>

2.10   Resolving Types

In this section we briefly describe an internal process by which Axiom determines a type to which two objects of possibly different types can be converted. We do this to give you further insight into how Axiom takes your input, analyzes it, and produces a result.

What happens when you enter $x+1$ to Axiom? Let's look at what you get from the two terms of this expression.

This is a symbolic object whose type indicates the name.

```
x
```

$$x$$

<div align="right">Type: Variable x</div>

This is a positive integer.

```
1
```

$$1$$

<div align="right">Type: PositiveInteger</div>

There are no operations in `PositiveInteger` that add positive integers to objects of type `Variable(x)` nor are there any in `Variable(x)`. Before it can add the two parts, Axiom must come up with a common type to which both x

and 1 can be converted. We say that Axiom must *resolve* the two types into a common type. In this example, the common type is `Polynomial(Integer)`.

Once this is determined, both parts are converted into polynomials, and the addition operation from `Polynomial(Integer)` is used to get the answer.

```
x + 1
```

$$x + 1$$

<div align="right">Type: Polynomial Integer</div>

Axiom can always resolve two types: if nothing resembling the original types can be found, then `Any` is be used. This is fine and useful in some cases.

```
["string",3.14159]
```

$$["string", 3.14159]$$

<div align="right">Type: List Any</div>

In other cases objects of type `Any` can't be used by the operations you specified.

```
"string" + 3.14159
```

```
There are 11 exposed and 5 unexposed library operations named +
  having 2 argument(s) but none was determined to be applicable.
  Use HyperDoc Browse, or issue
                          )display op +
  to learn more about the available operations. Perhaps
  package-calling the operation or using coercions on the
  arguments will allow you to apply the operation.

Cannot find a definition or applicable library operation named +
  with argument type(s)
                              String
                               Float

  Perhaps you should use "@" to indicate the required return type,
  or "$" to specify which version of the function you need.
```

Although this example was contrived, your expressions may need to be qualified slightly to help Axiom resolve the types involved. You may need to declare a few variables, do some package calling, provide some target type information or do some explicit conversions.

We suggest that you just enter the expression you want evaluated and see what Axiom does. We think you will be impressed with its ability to "do what I mean." If Axiom is still being obtuse, give it some hints. As you work with Axiom, you will learn where it needs a little help to analyze quickly and perform your computations.

2.11 Exposing Domains and Packages

In this section we discuss how Axiom makes some operations available to you while hiding others that are meant to be used by developers or only in rare cases. If you are a new user of Axiom, it is likely that everything you need is available by default and you may want to skip over this section on first reading.

Every domain and package in the Axiom library is either exposed (meaning that you can use its operations without doing anything special) or it is *hidden* (meaning you have to either package call (see 2.9 on page 162) the operations it contains or explicitly expose it to use the operations). The initial exposure status for a constructor is set in the file **exposed.lsp** (see the *Installer's Note* for Axiom if you need to know the location of this file). Constructors are collected together in *exposure groups*. Categories are all in the exposure group "categories" and the bulk of the basic set of packages and domains that are exposed are in the exposure group "basic." Here is an abbreviated sample of the file (without the Lisp parentheses):

```
basic
        AlgebraicNumber                         AN
        AlgebraGivenByStructuralConstants       ALGSC
        Any                                     ANY
        AnyFunctions1                           ANY1
        BinaryExpansion                         BINARY
        Boolean                                 BOOLEAN
        CardinalNumber                          CARD
        CartesianTensor                         CARTEN
        Character                               CHAR
        CharacterClass                          CCLASS
        CliffordAlgebra                         CLIF
        Color                                   COLOR
        Complex                                 COMPLEX
        ContinuedFraction                       CONTFRAC
        DecimalExpansion                        DECIMAL
        ...


categories
        AbelianGroup                            ABELGRP
        AbelianMonoid                           ABELMON
```

```
AbelianMonoidRing                        AMR
AbelianSemiGroup                         ABELSG
Aggregate                                AGG
Algebra                                  ALGEBRA
AlgebraicallyClosedField                 ACF
AlgebraicallyClosedFunctionSpace         ACFS
ArcHyperbolicFunctionCategory            AHYP
...
```

For each constructor in a group, the full name and the abbreviation is given. There are other groups in **exposed.lsp** but initially only the constructors in exposure groups "basic" "categories" "naglink" and "anna" are exposed.

As an interactive user of Axiom, you do not need to modify this file. Instead, use `)set expose` to expose, hide or query the exposure status of an individual constructor or exposure group. The reason for having exposure groups is to be able to expose or hide multiple constructors with a single command. For example, you might group together into exposure group "quantum" a number of domains and packages useful for quantum mechanical computations. These probably should not be available to every user, but you want an easy way to make the whole collection visible to Axiom when it is looking for operations to apply.

If you wanted to hide all the basic constructors available by default, you would issue `)set expose drop group basic`. We do not recommend that you do this. If, however, you discover that you have hidden all the basic constructors, you should issue `)set expose add group basic` to restore your default environment.

It is more likely that you would want to expose or hide individual constructors. In we use several operations from `OutputForm`, a domain usually hidden. To avoid package calling every operation from `OutputForm`, we expose the domain and let Axiom conclude that those operations should be used. Use `)set expose add constructor` and `)set expose drop constructor` to expose and hide a constructor, respectively. You should use the constructor name, not the abbreviation. The `)set expose` command guides you through these options.

If you expose a previously hidden constructor, Axiom exhibits new behavior (that was your intention) though you might not expect the results that you get. `OutputForm` is, in fact, one of the worst offenders in this regard. This domain is meant to be used by other domains for creating a structure that Axiom knows how to display. It has functions like "`+`" that form output representations rather than do mathematical calculations. Because of the order in which Axiom looks at constructors when it is deciding what operation to apply, `OutputForm` might be used instead of what you expect.

This is a polynomial.

```
x + x
```

$$2\,x$$

<div align="right">Type: Polynomial Integer</div>

Expose `OutputForm`.

```
)set expose add constructor OutputForm
```

```
   OutputForm is now explicitly exposed in frame G82322
```

This is what we get when `OutputForm` is automatically available.

```
x + x
```

$$x + x$$

<div align="right">Type: OutputForm</div>

Hide `OutputForm` so we don't run into problems with any later examples!

```
)set expose drop constructor OutputForm
```

```
   OutputForm is now explicitly hidden in frame G82322
```

Finally, exposure is done on a frame-by-frame basis. A *frame* (see 1.11 on page 1001) is one of possibly several logical Axiom workspaces within a physical one, each having its own environment (for example, variables and function definitions). If you have several Axiom workspace windows on your screen, they are all different frames, automatically created for you by HyperDoc. Frames can be manually created, made active and destroyed by the `)frame` system command. They do not share exposure information, so you need to use `)set expose` in each one to add or drop constructors from view.

2.12   Commands for Snooping

To conclude this chapter, we introduce you to some system commands that you can use for getting more information about domains, packages, categories, and operations. The most powerful Axiom facility for getting information about constructors and operations is the Browse component of HyperDoc. This is discussed in Chapter 14 on page 943.

Use the `)what` system command to see lists of system objects whose name contain a particular substring (uppercase or lowercase is not significant).

Issue this to see a list of all operations with "`complex`" in their names.

```
)what operation complex
```

```
Operations whose names satisfy the above pattern(s):
```

```
complex                     complex?
complexEigenvalues          complexEigenvectors
complexElementary           complexExpand
complexForm                 complexIntegrate
complexLimit                complexNormalize
complexNumeric              complexNumericIfCan
complexRoots                complexSolve
complexZeros                createLowComplexityNormalBasis
createLowComplexityTable    doubleComplex?
drawComplex                 drawComplexVectorField
fortranComplex              fortranDoubleComplex
pmComplexintegrate
```

```
To get more information about an operation such as
complexZeros, issue the command )display op complexZeros
```

If you want to see all domains with "`matrix`" in their names, issue this.

```
)what domain matrix
```

```
---------------------- Domains ----------------------
```

```
Domains with names matching patterns:
     matrix
```

```
 DHMATRIX DenavitHartenbergMatrix
 DPMM     DirectProductMatrixModule
 IMATRIX  IndexedMatrix
 LSQM     LieSquareMatrix
 M3D      ThreeDimensionalMatrix
 MATCAT-  MatrixCategory&
 MATRIX   Matrix
 RMATCAT- RectangularMatrixCategory&
 RMATRIX  RectangularMatrix
 SMATCAT- SquareMatrixCategory&
 SQMATRIX SquareMatrix
```

Similarly, if you wish to see all packages whose names contain "`gauss`", enter this.

```
)what package gauss
```

```
---------------------- Packages ----------------------
```

```
Packages with names matching patterns:
     gauss
```

```
 GAUSSFAC GaussianFactorizationPackage
```

This command shows all the operations that **Any** provides. Wherever **$** appears, it means "**Any**".

```
)show Any
```

```
 Any  is a domain constructor
 Abbreviation for Any is ANY
 This constructor is exposed in this frame.
 Issue )edit /usr/local/axiom/mnt/algebra/any.spad
  to see algebra source code for ANY
```

```
--------------------- Operations ---------------------
 ?=? : (%,%) -> Boolean
 any : (SExpression,None) -> %
 coerce : % -> OutputForm
 dom : % -> SExpression
 domainOf : % -> OutputForm
 hash : % -> SingleInteger
 latex : % -> String
 obj : % -> None
 objectOf : % -> OutputForm
 ?~=? : (%,%) -> Boolean
 showTypeInOutput : Boolean -> String
```

This displays all operations with the name **complex**.

```
)display operation complex
```

```
There is one exposed function called complex :
 [1] (D1,D1) -> D from D if D has COMPCAT D1 and D1 has COMRING
```

Let's analyze this output.

First we find out what some of the abbreviations mean.

```
)abbreviation query COMPCAT
```

```
   COMPCAT abbreviates category ComplexCategory
```

```
)abbreviation query COMRING
```

```
   COMRING abbreviates category CommutativeRing
```

So if `D1` is a commutative ring (such as the integers or floats) and `D` belongs
to `ComplexCategory D1`, then there is an operation called **complex** that takes
two elements of `D1` and creates an element of `D`. The primary example of a
constructor implementing domains belonging to `ComplexCategory` is `Complex`.
See **??** on page **??** for more information on that and see 6.4 on page 237 for
more information on function types.

Chapter 3

Using HyperDoc
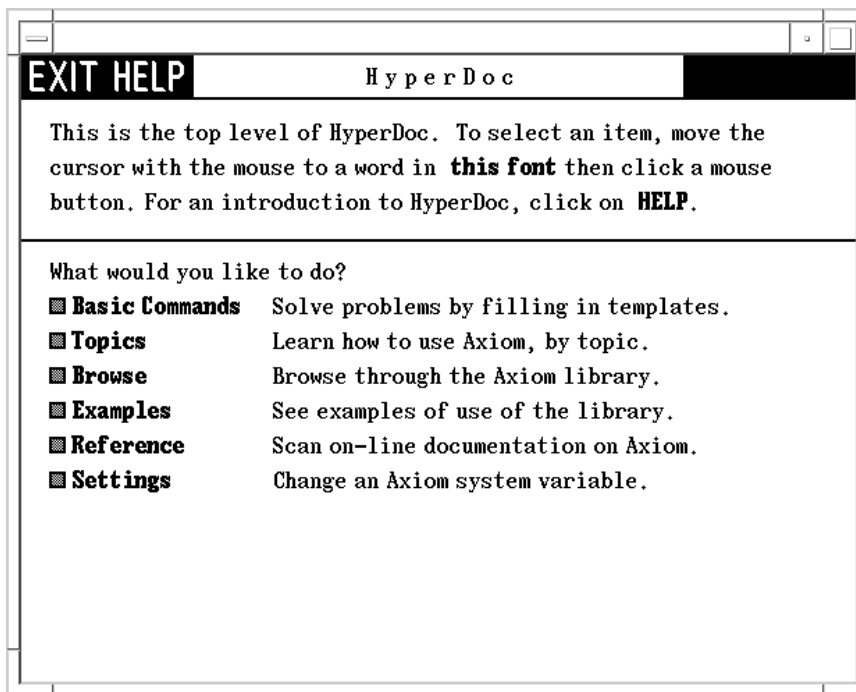
Figure 3.1: The HyperDoc root window page.

HyperDoc is the gateway to Axiom. It's both an on-line tutorial and an on-line reference manual. It also enables you to use Axiom simply by using the mouse and filling in templates. HyperDoc is available to you if you are running Axiom under the X Window System.

175

Pages usually have active areas, marked in **this font** (bold face). As you move the mouse pointer to an active area, the pointer changes from a filled dot to an open circle. The active areas are usually linked to other pages. When you click on an active area, you move to the linked page.

3.1   Headings

Most pages have a standard set of buttons at the top of the page. This is what they mean:

HELP Click on this to get help. The button only appears if there is specific help for the page you are viewing. You can get *general* help for HyperDoc by clicking the help button on the home page.

⬅ Click here to go back one page. By clicking on this button repeatedly, you can go back several pages and then take off in a new direction.

HOME Go back to the home page, that is, the page on which you started. Use HyperDoc to explore, to make forays into new topics. Don't worry about how to get back. HyperDoc remembers where you came from. Just click on this button to return.

EXIT From the root window (the one that is displayed when you start the system) this button leaves the HyperDoc program, and it must be restarted if you want to use it again. From any other HyperDoc window, it just makes that one window go away. You *must* use this button to get rid of a window. If you use the window manager "Close" button, then all of HyperDoc goes away.

The buttons are not displayed if they are not applicable to the page you are viewing. For example, there is no **HOME** button on the top-level menu.

3.2   Key Definitions

The following keyboard definitions are in effect throughout HyperDoc. See 3.3 on page 177 and 3.4 on page 178 for some contextual key definitions.

F1 Display the main help page.

F3 Same as **EXIT**, makes the window go away if you are not at the top-level window or quits the HyperDoc facility if you are at the top-level.

F5 Rereads the HyperDoc database, if necessary (for system developers).

F9 Displays this information about key definitions.

F12 Same as **F3**.

Up Arrow Scroll up one line.

Down Arrow Scroll down one line.

Page Up Scroll up one page.

Page Down Scroll down one page.

3.3 Scroll Bars

Whenever there is too much text to fit on a page, a *scroll bar* automatically appears along the right side.

With a scroll bar, your page becomes an aperture, that is, a window into a larger amount of text than can be displayed at one time. The scroll bar lets you move up and down in the text to see different parts. It also shows where the aperture is relative to the whole text. The aperture is indicated by a strip on the scroll bar.

Move the cursor with the mouse to the "down-arrow" at the bottom of the scroll bar and click. See that the aperture moves down one line. Do it several times. Each time you click, the aperture moves down one line. Move the mouse to the "up-arrow" at the top of the scroll bar and click. The aperture moves up one line each time you click.

Next move the mouse to any position along the middle of the scroll bar and click. HyperDoc attempts to move the top of the aperture to this point in the text.

You cannot make the aperture go off the bottom edge. When the aperture is about half the size of text, the lowest you can move the aperture is halfway down.

To move up or down one screen at a time, use the $\boxed{\textbf{PageUp}}$ and $\boxed{\textbf{PageDown}}$ keys on your keyboard. They move the visible part of the region up and down one page each time you press them.

If the HyperDoc page does not contain an input area (see 3.4 on page 178), you can also use the $\boxed{\textbf{Home}}$ and $\boxed{\uparrow}$ and $\boxed{\downarrow}$ arrow keys to navigate. When you press the $\boxed{\textbf{Home}}$ key, the screen is positioned at the very top of the page. Use the $\boxed{\uparrow}$ and $\boxed{\downarrow}$ arrow keys to move the screen up and down one line at a time, respectively.

3.4   Input Areas

Input areas are boxes where you can put data.

To enter characters, first move your mouse cursor to somewhere within the HyperDoc page. Characters that you type are inserted in front of the underscore. This means that when you type characters at your keyboard, they go into this first input area.

The input area grows to accommodate as many characters as you type. Use the $\boxed{\textbf{Backspace}}$ key to erase characters to the left. To modify what you type, use the right-arrow $\boxed{\rightarrow}$ and left-arrow keys $\boxed{\leftarrow}$ and the keys $\boxed{\textbf{Insert}}$, $\boxed{\textbf{Delete}}$, $\boxed{\textbf{Home}}$ and $\boxed{\textbf{End}}$. These keys are found immediately on the right of the standard IBM keyboard.

If you press the $\boxed{\textbf{Home}}$ key, the cursor moves to the beginning of the line and if you press the $\boxed{\textbf{End}}$ key, the cursor moves to the end of the line. Pressing $\boxed{\textbf{Ctrl}}$–$\boxed{\textbf{End}}$ deletes all the text from the cursor to the end of the line.

A page may have more than one input area. Only one input area has an underscore cursor. When you first see apage, the top-most input area contains the cursor. To type information into another input area, use the $\boxed{\textbf{Enter}}$ or $\boxed{\textbf{Tab}}$ key to move from one input area to xanother. To move in the reverse order, use $\boxed{\textbf{Shift}}$–$\boxed{\textbf{Tab}}$.

You can also move from one input area to another using your mouse. Notice that each input area is active. Click on one of the areas. As you can see, the underscore cursor moves to that window.

3.5   Radio Buttons and Toggles

Some pages have *radio buttons* and *toggles*. Radio buttons are a group of buttons like those on car radios: you can select only one at a time.

Once you have selected a button, it appears to be inverted and contains a checkmark. To change the selection, move the cursor with the mouse to a different radio button and click.

A toggle is an independent button that displays some on/off state. When "on", the button appears to be inverted and contains a checkmark. When "off", the button is raised.

Unlike radio buttons, you can set a group of them any way you like. To change toggle the selection, move the cursor with the mouse to the button and click.

3.6 Search Strings

A *search string* is used for searching some database. To learn about search strings, we suggest that you bring up the HyperDoc glossary. To do this from the top-level page of HyperDoc:

1. Click on Reference, bringing up the Axiom Reference page.

2. Click on Glossary, bringing up the glossary.

The glossary has an input area at its bottom. We review the various kinds of search strings you can enter to search the glossary.

The simplest search string is a word, for example, `operation`. A word only matches an entry having exactly that spelling. Enter the word `operation` into the input area above then click on **Search**. As you can see, `operation` matches only one entry, namely with `operation` itself.

Normally matching is insensitive to whether the alphabetic characters of your search string are in uppercase or lowercase. Thus `operation` and `OperAtion` both have the same effect.

You will very often want to use the wildcard "`*`" in your search string so as to match multiple entries in the list. The search key "`*`" matches every entry in the list. You can also use "`*`" anywhere within a search string to match an arbitrary substring. Try "`cat*`" for example: enter "`cat*`" into the input area and click on **Search**. This matches several entries.

You use any number of wildcards in a search string as long as they are not adjacent. Try search strings such as "`*dom*`". As you see, this search string matches "`domain`", "`domain constructor`", "`subdomain`", and so on.

3.6.1 Logical Searches

For more complicated searches, you can use "`and`", "`or`", and "`not`" with basic search strings; write logical expressions using these three operators just as in the Axiom language. For example, `domain or package` matches the two entries `domain` and `package`. Similarly, "`dom* and *con*`" matches "`domain constructor`" and others. Also "`not *a*`" matches every entry that does not contain the letter "`a`" somewhere.

Use parentheses for grouping. For example, "`dom* and (not *con*)`" matches "`domain`" but not "`domain constructor`".

There is no limit to how complex your logical expression can be. For example,

```
a* or b* or c* or d* or e* and (not *a*)
```

is a valid expression.

3.7   Example Pages

Many pages have Axiom example commands.

Each command has an active "button" along the left margin. When you click on this button, the output for the command is "pasted-in." Click again on the button and you see that the pasted-in output disappears.

Maybe you would like to run an example? To do so, just click on any part of its text! When you do, the example line is copied into a new interactive Axiom buffer for this HyperDoc page.

Sometimes one example line cannot be run before you run an earlier one. Don't worry—HyperDoc automatically runs all the necessary lines in the right order!

The new interactive Axiom buffer disappears when you leave HyperDoc. If you want to get rid of it beforehand, use the **Cancel** button of the X Window manager or issue the Axiom system command `)close.`

3.8   X Window Resources for HyperDoc

You can control the appearance of HyperDoc while running under Version 11 of the X Window System by placing the following resources in the file **.Xdefaults** in your home directory. In what follows, *font* is any valid X11 font name (for example, `Rom14`) and *color* is any valid X11 color specification (for example, `NavyBlue`). For more information about fonts and colors, refer to the X Window documentation for your system.

`Axiom.hyperdoc.RmFont:` *font*
    This is the standard text font. The default value is `Rom14`

`Axiom.hyperdoc.RmColor:` *color*
    This is the standard text color. The default value is `black`

`Axiom.hyperdoc.ActiveFont:` *font*
    This is the font used for HyperDoc link buttons. The default value is `Bld14`

`Axiom.hyperdoc.ActiveColor:` *color*
    This is the color used for HyperDoc link buttons. The default value is `black`

`Axiom.hyperdoc.AxiomFont:` *font*
    This is the font used for active Axiom commands. The default value is `Bld14`

`Axiom.hyperdoc.AxiomColor:` *color*
    This is the color used for active Axiom commands. The default value is `black`

`Axiom.hyperdoc.BoldFont:` *font*

> This is the font used for bold face. The default value is `Bld14`

`Axiom.hyperdoc.BoldColor:` *color*

> This is the color used for bold face. The default value is `black`

`Axiom.hyperdoc.TtFont:` *font*

> This is the font used for Axiom output in HyperDoc. This font must be fixed-width. The default value is `Rom14`

`Axiom.hyperdoc.TtColor:` *color*

> This is the color used for Axiom output in HyperDoc. The default value is `black`

`Axiom.hyperdoc.EmphasizeFont:` *font*

> This is the font used for italics. The default value is `Itl14`

`Axiom.hyperdoc.EmphasizeColor:` *color*

> This is the color used for italics. The default value is `black`

`Axiom.hyperdoc.InputBackground:` *color*

> This is the color used as the background for input areas. The default value is `black`

`Axiom.hyperdoc.InputForeground:` *color*

> This is the color used as the foreground for input areas. The default value is `white`

`Axiom.hyperdoc.BorderColor:` *color*

> This is the color used for drawing border lines. The default value is `black`

`Axiom.hyperdoc.Background:` *color*

> This is the color used for the background of all windows. The default value is `white`

Chapter 4

Input Files and Output Styles

In this chapter we discuss how to collect Axiom statements and commands into files and then read the contents into the workspace. We also show how to display the results of your computations in several different styles including TeX, FORTRAN and monospace two-dimensional format.[1]

The printed version of this book uses the Axiom TeX output formatter. When we demonstrate a particular output style, we will need to turn TeX formatting off and the output style on so that the correct output is shown in the text.

4.1 Input Files

In this section we explain what an *input file* is and why you would want to know about it. We discuss where Axiom looks for input files and how you can direct it to look elsewhere. We also show how to read the contents of an input file into the *workspace* and how to use the *history* facility to generate an input file from the statements you have entered directly into the workspace.

An *input* file contains Axiom expressions and system commands. Anything that you can enter directly to Axiom can be put into an input file. This is how you save input functions and expressions that you wish to read into Axiom more than one time.

To read an input file into Axiom, use the `)read` system command. For example, you can read a file in a particular directory by issuing

```
)read /spad/src/input/matrix.input
```

---

[1] TeX is a trademark of the American Mathematical Society.

The ".**input**" is optional; this also works:

```
)read /spad/src/input/matrix
```

What happens if you just enter `)read matrix.input` or even `)read matrix`?
Axiom looks in your current working directory for input files that are not qual-
ified by a directory name. Typically, this directory is the directory from which
you invoked Axiom.

To change the current working directory, use the `)cd` system command. The
command `)cd` by itself shows the current working directory. To change it to the
src/input subdirectory for user "babar", issue

```
)cd /u/babar/src/input
```

Axiom looks first in this directory for an input file. If it is not found, it looks in
the system's directories, assuming you meant some input file that was provided
with Axiom.

> If you have the Axiom history facility turned on (which it is by default), you
> can save all the lines you have entered into the workspace by entering
>)history)write
> Axiom tells you what input file to edit to see your statements. The file is in
> your home directory or in the directory you specified with `)cd`.

In 5.2 on page 199 we discuss using indentation in input files to group statements
into *blocks*.

4.2   The .axiom.input File

When Axiom starts up, it tries to read the input file **.axiom.input**[2] from your
home directory. It there is no **.axiom.input** in your home directory, it reads
the copy located in its own **src/input** directory. The file usually contains
system commands to personalize your Axiom environment. In the remainder
of this section we mention a few things that users frequently place in their
.axiom.input files.

In order to have FORTRAN output always produced from your computations,
place the system command `)set output fortran on` in **.axiom.input**. If you
do not want to be prompted for confirmation when you issue the `)quit` system
command, place `)set quit unprotected` in **.axiom.input**. If you then decide
that you do want to be prompted, issue `)set quit protected`. This is the
default setting so that new users do not leave Axiom inadvertently.[3]

---

[2]**.axiom.input** used to be called **axiom.input** in the NAG version

[3]The system command `)pquit` always prompts you for confirmation.

To see the other system variables you can set, issue `)set` or use the HyperDoc
Settings facility to view and change Axiom system variables.

4.3 Common Features of Using Output Formats

In this section we discuss how to start and stop the display of the different
output formats and how to send the output to the screen or to a file. To fix
ideas, we use FORTRAN output format for most of the examples.

You can use the `)set output` system command to toggle or redirect the differ-
ent kinds of output. The name of the kind of output follows "output" in the
command. The names are

fortran   for FORTRAN output.
algebra   for monospace two-dimensional mathematical output.
tex   for TeX output.
script   for IBM Script Formula Format output.

For example, issue `)set output fortran on` to turn on FORTRAN format
and issue `)set output fortran off` to turn it off. By default, `algebra` is `on`
and all others are `off`. When output is started, it is sent to the screen. To send
the output to a file, give the file name without directory or extension. Axiom
appends a file extension depending on the kind of output being produced.

Issue this to redirect FORTRAN output to, for example, the file **linalg.sfort**.

```
)set output fortran linalg
```

```
   FORTRAN output will be written to file linalg.sfort .
```

You must *also* turn on the creation of FORTRAN output. The above just says
where it goes if it is created.

```
)set output fortran on
```

In what directory is this output placed? It goes into the directory from which
you started Axiom, or if you have used the `)cd` system command, the one that
you specified with `)cd`. You should use `)cd` before you send the output to the
file.

You can always direct output back to the screen by issuing this.

```
)set output fortran console
```

Let's make sure FORTRAN formatting is off so that nothing we do from now
on produces FORTRAN output.

```
)set output fortran off
```

We also delete the demonstrated output file we created.

```
)system rm linalg.sfort
```

You can abbreviate the words "`on`," "`off`," and "`console`" to the minimal number of characters needed to distinguish them. Because of this, you cannot send output to files called **on.sfort, off.sfort, of.sfort, console.sfort, consol.sfort** and so on.

The width of the output on the page is set by `)set output length` for all formats except FORTRAN. Use `)set fortran fortlength` to change the FORTRAN line length from its default value of 72.

4.4   Monospace Two-Dimensional Mathematical Format

This is the default output format for Axiom. It is usually on when you start the system.

If it is not, issue this.

```
)set output algebra on
```

Since the printed version of this book (as opposed to the HyperDoc version) shows output produced by the TEX output formatter, let us temporarily turn off TEX output.

```
)set output tex off
```

Here is an example of what it looks like.

```
matrix [ [i*x**i + j*%i*y**j for i in 1..2] for j in 3..4]
```

```
        +      3            3     2+
        |3%i y  + x   3%i y  + 2x |
   (1)  |                         |
        |     4            4     2|
        +4%i y  + x   4%i y  + 2x +
```

                          Type: Matrix Polynomial Complex Integer

Issue this to turn off this kind of formatting.

```
)set output algebra off
```

Turn TEX output on again.

```
)set output tex on
```

The characters used for the matrix brackets above are rather ugly. You get this character set when you issue `)set output characters plain`. This character set should be used when you are running on a machine that does not support the IBM extended ASCII character set. If you are running on an IBM workstation, for example, issue `)set output characters default` to get better looking output.

4.5   TeX Format

Axiom can produce TEX output for your expressions. The output is produced using macros from the LATEX document preparation system by Leslie Lamport[1]. The printed version of this book was produced using this formatter.

To turn on TEX output formatting, issue this.

```
)set output tex on
```

Here is an example of its output.

```
matrix [ [i*x**i + j*\%i*y**j for i in 1..2] for j in 3..4]
```

```
$$
\left[
\begin{array}{cc}
{{3 \  i \  {y \sp 3}}+x} &
{{3 \  i \  {y \sp 3}}+{2 \  {x \sp 2}}} \\
{{4 \  i \  {y \sp 4}}+x} &
{{4 \  i \  {y \sp 4}}+{2 \  {x \sp 2}}}
\end{array}
\right]
$$
```

This formats as

$$
\left[
\begin{array}{cc}
3\ i\ y^3 + x & 3\ i\ y^3 + 2\ x^2 \\
4\ i\ y^4 + x & 4\ i\ y^4 + 2\ x^2
\end{array}
\right]
$$

To turn TeX output formatting off, issue `)set output tex off`. The LaTeXmacros in the output generated by Axiom are all standard except for the following definitions:

```
\def\csch{\mathop{\rm csch}\nolimits}
```

```
\def\erf{\mathop{\rm erf}\nolimits}
```

```
\def\zag#1#2{
  {{\hfill \left. {#1} \right|}
   \over
   {\left| {#2} \right. \hfill}
  }
}
```

4.6   IBM Script Formula Format

Axiom can produce IBM Script Formula Format output for your expressions.

To turn IBM Script Formula Format on, issue this.

```
)set output script on
```

Here is an example of its output.

```
matrix [ [i*x**i + j*%i*y**j for i in 1..2] for j in 3..4]
```

```
.eq set blank @
:df.
<left lb < < < <3 @@ %i @@ <y sup 3> >+x> here < <3 @@ %i @@
<y sup 3> >+<2 @@ <x sup 2> > > > habove < < <4 @@ %i @@
<y sup 4> >+x> here < <4 @@ %i @@ <y sup 4> >+<2 @@
<x up 2> > > > > right rb>
:edf.
```

To turn IBM Script Formula Format output formatting off, issue this.

```
)set output script off
```

4.7   FORTRAN Format

In addition to turning FORTRAN output on and off and stating where the output should be placed, there are many options that control the appearance of

the generated code. In this section we describe some of the basic options. Issue `)set fortran` to see a full list with their current settings.

The output FORTRAN expression usually begins in column 7. If the expression needs more than one line, the ampersand character `&` is used in column 6. Since some versions of FORTRAN have restrictions on the number of lines per statement, Axiom breaks long expressions into segments with a maximum of 1320 characters (20 lines of 66 characters) per segment. If you want to change this, say, to 660 characters, issue the system command `)set fortran explength 660`. You can turn off the line breaking by issuing `)set fortran segment off`. Various code optimization levels are available.

FORTRAN output is produced after you issue this.

```
)set output fortran on
```

For the initial examples, we set the optimization level to 0, which is the lowest level.

```
)set fortran optlevel 0
```

The output is usually in columns 7 through 72, although fewer columns are used in the following examples so that the output fits nicely on the page.

```
)set fortran fortlength 60
```

By default, the output goes to the screen and is displayed before the standard Axiom two-dimensional output. In this example, an assignment to the variable $R1$ was generated because this is the result of step 1.

```
(x+y)**3
```

```
      R1=y**3+3*x*y*y+3*x*x*y+x**3
```

$$y^3 + 3 \ x \ y^2 + 3 \ x^2 \ y + x^3$$

```
                              Type:  Polynomial Integer
```

Here is an example that illustrates the line breaking.

```
(x+y+z)**3
```

```
      R2=z**3+(3*y+3*x)*z*z+(3*y*y+6*x*y+3*x*x)*z+y**3+3*x*y
     &*y+3*x*x*y+x**3
```

$$z^3 + \left(3\ y + 3\ x\right)\ z^2 + \left(3\ y^2 + 6\ x\ y + 3\ x^2\right)\ z + y^3 + 3\ x\ y^2 + 3\ x^2\ y + x^3$$

```
                              Type:  Polynomial Integer
```

Note in the above examples that integers are generally converted to floating point numbers, except in exponents. This is the default behavior but can be turned off by issuing `)set fortran ints2floats off`. The rules governing when the conversion is done are:

1. If an integer is an exponent, convert it to a floating point number if it is greater than 32767 in absolute value, otherwise leave it as an integer.

2. Convert all other integers in an expression to floating point numbers.

These rules only govern integers in expressions. Numbers generated by Axiom for *DIMENSION* statements are also integers.

To set the type of generated FORTRAN data, use one of the following:

```
)set fortran defaulttype REAL
)set fortran defaulttype INTEGER
)set fortran defaulttype COMPLEX
)set fortran defaulttype LOGICAL
)set fortran defaulttype CHARACTER
```

When temporaries are created, they are given a default type of `REAL`. Also, the `REAL` versions of functions are used by default.

```
sin(x)
```

```
    R3=DSIN(x)
```

$$\sin\left(x\right)$$

```
                              Type:  Expression Integer
```

At optimization level 1, Axiom removes common subexpressions.

```
)set fortran optlevel 1
```

```
(x+y+z)**3
```

```
    T2=y*y
    T3=x*x
    R4=z**3+(3*y+3*x)*z*z+(3*T2+6*x*y+3*T3)*z+y**3+3*x*T2+
    &3*T3*y+x**3
```

$$z^3 + (3\ y + 3\ x)\ z^2 + \left(3\ y^2 + 6\ x\ y + 3\ x^2\right)\ z + y^3 + 3\ x\ y^2 + 3\ x^2\ y + x^3$$

<div align="right">Type: Polynomial Integer</div>

This changes the precision to `DOUBLE`. Substitute `single` for `double` to return to single precision.

```
)set fortran precision double
```

Complex constants display the precision.

```
2.3 + 5.6*%i
```

```
    R5=(2.3D0,5.6D0)
```

$$2.3 + 5.6\ i$$

<div align="right">Type: Complex Float</div>

The function names that Axiom generates depend on the chosen precision.

```
sin %e
```

```
    R6=DSIN(DEXP(1))
```

$$\sin{(e)}$$

<div align="right">Type: Expression Integer</div>

Reset the precision to `single` and look at these two examples again.

```
)set fortran precision single
```

```
2.3 + 5.6*%i
```

```
    R7=(2.3,5.6)
```

$$2.3 + 5.6\ i$$

<div align="right">Type: Complex Float</div>

```
sin %e
```

```
      R8=SIN(EXP(1))
```

$$\sin{(e)}$$

<div align="right">Type: Expression Integer</div>

Expressions that look like lists, streams, sets or matrices cause array code to be generated.

```
[x+1,y+1,z+1]
```

```
      T1(1)=x+1
      T1(2)=y+1
      T1(3)=z+1
      R9=T1
```

$$[x+1, y+1, z+1]$$

<div align="right">Type: List Polynomial Integer</div>

A temporary variable is generated to be the name of the array. This may have to be changed in your particular application.

```
set[2,3,4,3,5]
```

```
      T1(1)=2
      T1(2)=3
      T1(3)=4
      T1(4)=5
      R10=T1
```

$$\{2, 3, 4, 5\}$$

<div align="right">Type: Set PositiveInteger</div>

By default, the starting index for generated FORTRAN arrays is 0.

```
matrix [ [2.3,9.7],[0.0,18.778] ]
```

```
T1(0,0)=2.3
T1(0,1)=9.7
T1(1,0)=0.0
T1(1,1)=18.778
T1
```

$$\begin{bmatrix} 2.3 & 9.7 \\ 0.0 & 18.778 \end{bmatrix}$$

Type: Matrix Float

To change the starting index for generated FORTRAN arrays to be 1, issue this. This value can only be 0 or 1.

```
)set fortran startindex 1
```

Look at the code generated for the matrix again.

```
matrix [ [2.3,9.7],[0.0,18.778] ]
```

```
T1(1,1)=2.3
T1(1,2)=9.7
T1(2,1)=0.0
T1(2,2)=18.778
T1
```

$$\begin{bmatrix} 2.3 & 9.7 \\ 0.0 & 18.778 \end{bmatrix}$$

Type: Matrix Float

Chapter 5

Overview of Interactive Language

In this chapter we look at some of the basic components of the Axiom language that you can use interactively. We show how to create a *block* of expressions, how to form loops and list iterations, how to modify the sequential evaluation of a block and how to use `if-then-else` to evaluate parts of your program conditionally. We suggest you first read the boxed material in each section and then proceed to a more thorough reading of the chapter.

5.1   Immediate and Delayed Assignments

A *variable* in Axiom refers to a value. A variable has a name beginning with an uppercase or lowercase alphabetic character, "%", or "!". Successive characters (if any) can be any of the above, digits, or "?". Case is distinguished. The following are all examples of valid, distinct variable names:

```
a            tooBig?    a1B2c3%!?
A            %j         numberOfPoints
beta6        %J         numberofpoints
```

The ":=" operator is the immediate *assignment* operator. Use it to associate a value with a variable.

The syntax for immediate assignment for a single variable is

$$variable := expression$$

The value returned by an immediate assignment is the value of *expression*.

The right-hand side of the expression is evaluated, yielding 1. This value is then assigned to a.

```
a := 1
```

$$1$$

                                              Type: PositiveInteger

The right-hand side of the expression is evaluated, yielding 1. This value is then assigned to b. Thus a and b both have the value 1 after the sequence of assignments.

```
b := a
```

$$1$$

                                              Type: PositiveInteger

What is the value of b if a is assigned the value 2?

```
a := 2
```

$$2$$

                                              Type: PositiveInteger

As you see, the value of b is left unchanged.

```
b
```

$$1$$

                                              Type: PositiveInteger

This is what we mean when we say this kind of assignment is *immediate*; b has no dependency on a after the initial assignment. This is the usual notion of assignment found in programming languages such as C, PASCAL and FOR-TRAN.

Axiom provides delayed assignment with "==". This implements a delayed evaluation of the right-hand side and dependency checking.

> The syntax for delayed assignment is
>
> $$variable == expression$$
>
> The value returned by a delayed assignment is the unique value of `Void`.

Using a and b as above, these are the corresponding delayed assignments.

```
a == 1
```

                                                          Type: Void

```
b == a
```

                                                          Type: Void

The right-hand side of each delayed assignment is left unevaluated until the variables on the left-hand sides are evaluated. Therefore this evaluation and ...

```
a
```

```
Compiling body of rule a to compute value of type PositiveInteger
```

                             1

                                          Type: PositiveInteger

this evaluation seem the same as before.

```
b
```

```
Compiling body of rule b to compute value of type PositiveInteger
```

                             1

                                          Type: PositiveInteger

If we change a to 2

```
a == 2
```

```
   Compiled code for a has been cleared.
   Compiled code for b has been cleared.
   1 old definition(s) deleted for function or rule a
```

<div align="right">

Type: Void

</div>

then a evaluates to 2, as expected, but

```
a
```

```
Compiling body of rule a to compute value of type PositiveInteger
```

```
+++ |*0;a;1;G82322| redefined
```

<div align="center">

2

</div>

<div align="right">

Type: PositiveInteger

</div>

the value of b reflects the change to a.

```
b
```

```
Compiling body of rule b to compute value of type PositiveInteger
```

```
+++ |*0;b;1;G82322| redefined
```

<div align="center">

2

</div>

<div align="right">

Type: PositiveInteger

</div>

It is possible to set several variables at the same time by using a *tuple* of variables and a tuple of expressions. Note that a *tuple* is a collection of things separated by commas, often surrounded by parentheses.

The syntax for multiple immediate assignments is

$$(\mathit{var}_1, \ \mathit{var}_2, \ \ldots, \ \mathit{var}_N \ ) \ := \ (\ \mathit{expr}_1, \ \mathit{expr}_2, \ \ldots, \ \mathit{expr}_N \ )$$

The value returned by an immediate assignment is the value of expr_N.

This sets x to 1 and y to 2.

```
(x,y) := (1,2)
```

$$2$$

<div align="right">Type: PositiveInteger</div>

Multiple immediate assigments are parallel in the sense that the expressions on the right are all evaluated before any assignments on the left are made. However, the order of evaluation of these expressions is undefined.

You can use multiple immediate assignment to swap the values held by variables.

```
(x,y) := (y,x)
```

$$1$$

<div align="right">Type: PositiveInteger</div>

x has the previous value of y.

```
x
```

$$2$$

<div align="right">Type: PositiveInteger</div>

y has the previous value of x.

```
y
```

$$1$$

<div align="right">Type: PositiveInteger</div>

There is no syntactic form for multiple delayed assignments. See the discussion in section 6.8 on page 243 about how Axiom differentiates between delayed assignments and user functions of no arguments.

5.2 Blocks

A *block* is a sequence of expressions evaluated in the order that they appear, except as modified by control expressions such as `break`, `return`, `iterate` and `if-then-else` constructions. The value of a block is the value of the expression last evaluated in the block.

To leave a block early, use "=>". For example, $i < 0 => x$. The expression before the "=>" must evaluate to `true` or `false`. The expression following the "=>" is the return value for the block.

A block can be constructed in two ways:

1. the expressions can be separated by semicolons and the resulting expression surrounded by parentheses, and

2. the expressions can be written on succeeding lines with each line indented the same number of spaces (which must be greater than zero). A block entered in this form is called a *pile*.

Only the first form is available if you are entering expressions directly to Axiom. Both forms are available in **.input** files.

The syntax for a simple block of expressions entered interactively is

$$(\ expression_1; \ expression_2; \ \ldots; \ expression_N \ )$$

The value returned by a block is the value of an => expression, or $expression_N$ if no => is encountered.

In **.input** files, blocks can also be written using piles. The examples throughout this book are assumed to come from **.input** files.

In this example, we assign a rational number to a using a block consisting of three expressions. This block is written as a pile. Each expression in the pile has the same indentation, in this case two spaces to the right of the first line.

```
a :=
  i := gcd(234,672)
  i := 3*i**5 - i + 1
  1 / i
```

$$\frac{1}{23323}$$

```
                              Type:  Fraction Integer
```

Here is the same block written on one line. This is how you are required to enter it at the input prompt.

```
a := (i := gcd(234,672); i := 3*i**5 - i + 1; 1 / i)
```

$$\frac{1}{23323}$$

```
                              Type:  Fraction Integer
```

Blocks can be used to put several expressions on one line. The value returned is that of the last expression.

```
(a := 1; b := 2; c := 3; [a,b,c])
```

$$[1, 2, 3]$$

<div align="right">Type: List PositiveInteger</div>

Axiom gives you two ways of writing a block and the preferred way in an **.input** file is to use a pile. Roughly speaking, a pile is a block whose constituent expressions are indented the same amount. You begin a pile by starting a new line for the first expression, indenting it to the right of the previous line. You then enter the second expression on a new line, vertically aligning it with the first line. And so on. If you need to enter an inner pile, further indent its lines to the right of the outer pile. Axiom knows where a pile ends. It ends when a subsequent line is indented to the left of the pile or the end of the file.

Blocks can be used to perform several steps before an assignment (immediate or delayed) is made.

```
d :=
   c := a**2 + b**2
   sqrt(c * 1.3)
```

$$2.5495097567963924145$$

<div align="right">Type: Float</div>

Blocks can be used in the arguments to functions. (Here h is assigned $2.1 + 3.5$.)

```
h := 2.1 +
   1.0
   3.5
```

$$5.6$$

<div align="right">Type: Float</div>

Here the second argument to **eval** is $x = z$, where the value of z is computed in the first line of the block starting on the second line.

```
eval(x**2 - x*y**2,
   z := %pi/2.0 - exp(4.1)
   x = z
   )
```

$$58.769491270567072878 \ y^2 + 3453.853104201259382$$

                                                    Type: Polynomial Float

Blocks can be used in the clauses of `if-then-else` expressions (see 5.3 on page).

```
if h > 3.1 then 1.0 else (z := cos(h); max(z,0.5))
```

                                        1.0

                                                    Type: Float

This is the pile version of the last block.

```
if h > 3.1 then
    1.0
  else
    z := cos(h)
    max(z,0.5)
```

                                        1.0

                                                    Type: Float

Blocks can be nested.

```
a := (b := factorial(12); c := (d := eulerPhi(22);
factorial(d));b+c)
```

                                    482630400

                                                    Type: PositiveInteger

This is the pile version of the last block.

```
a :=
  b := factorial(12)
  c :=
    d := eulerPhi(22)
    factorial(d)
  b+c
```

                                    482630400

                                                    Type: PositiveInteger

Since $c + d$ does equal 3628855, a has the value of c and the last line is never evaluated.

```
a :=
  c := factorial 10
  d := fibonacci 10
  c + d = 3628855 => c
  d
```

$$3628800$$

$$\text{Type: PositiveInteger}$$

5.3 if-then-else

Like many other programming languages, Axiom uses the three keywords `if`, `then` and `else` to form conditional expressions. The `else` part of the conditional is optional. The expression between the `if` and `then` keywords is a *predicate*: an expression that evaluates to or is convertible to either `true` or `false`, that is, a `Boolean`.

The syntax for conditional expressions is

$$\text{if } predicate \text{ then } expression_1 \text{ else } expression_2$$

where the `else` $expression_2$ part is optional. The value returned from a conditional expression is $expression_1$ if the predicate evaluates to `true` and $expression_2$ otherwise. If no `else` clause is given, the value is always the unique value of `Void`.

An `if-then-else` expression always returns a value. If the `else` clause is missing then the entire expression returns the unique value of `Void`. If both clauses are present, the type of the value returned by `if` is obtained by resolving the types of the values of the two clauses. See 2.10 on page 166 for more information.

The predicate must evaluate to, or be convertible to, an object of type `Boolean`: `true` or `false`. By default, the equal sign "=" creates an equation.

This is an equation. In particular, it is an object of type `Equation Polynomial Integer`.

```
x + 1 = y
```

$$x + 1 = y$$

```
                                  Type:  Equation Polynomial Integer
```

However, for predicates in `if` expressions, Axiom places a default target type of `Boolean` on the predicate and equality testing is performed. Thus you need not qualify the "=" in any way. In other contexts you may need to tell Axiom that you want to test for equality rather than create an equation. In those cases, use "`@`" and a target type of `Boolean`. See section 2.9 on page 162 for more information.

The compound symbol meaning "not equal" in Axiom is "$\sim=$". This can be used directly without a package call or a target specification. The expression $a \sim= b$ is directly translated into $\mathtt{not}(a = b)$.

Many other functions have return values of type `Boolean`. These include "`<`", "`<=`", "`>`", "`>=`", "$\sim=$" and "**member?**". By convention, operations with names ending in "?" return `Boolean` values.

The usual rules for piles are suspended for conditional expressions. In **.input** files, the `then` and `else` keywords can begin in the same column as the corresponding `if` but may also appear to the right. Each of the following styles of writing `if-then-else` expressions is acceptable:

```
if i>0 then output("positive") else output("nonpositive")

if i > 0 then output("positive")
  else output("nonpositive")

if i > 0 then output("positive")
else output("nonpositive")

if i > 0
then output("positive")
else output("nonpositive")

if i > 0
  then output("positive")
  else output("nonpositive")
```

A block can follow the `then` or `else` keywords. In the following two assignments to `a`, the `then` and `else` clauses each are followed by two-line piles. The value returned in each is the value of the second line.

```
a :=
  if i > 0 then
    j := sin(i * pi())
    exp(j + 1/j)
  else
    j := cos(i * 0.5 * pi())
```

```
    log(abs(j)**5 + 1)

a :=
  if i > 0
    then
      j := sin(i * pi())
      exp(j + 1/j)
    else
      j := cos(i * 0.5 * pi())
      log(abs(j)**5 + 1)
```

These are both equivalent to the following:

```
a :=
  if i > 0 then (j := sin(i * pi()); exp(j + 1/j))
  else (j := cos(i * 0.5 * pi()); log(abs(j)**5 + 1))
```

5.4 Loops

A *loop* is an expression that contains another expression, called the *loop body*, which is to be evaluated zero or more times. All loops contain the `repeat` keyword and return the unique value of `Void`. Loops can contain inner loops to any depth.

The most basic loop is of the form

$$\texttt{repeat}\ loopBody$$

Unless *loopBody* contains a `break` or `return` expression, the loop repeats forever. The value returned by the loop is the unique value of `Void`.

5.4.1 Compiling vs. Interpreting Loops

Axiom tries to determine completely the type of every object in a loop and then to translate the loop body to LISP or even to machine code. This translation is called compilation.

If Axiom decides that it cannot compile the loop, it issues a message stating the problem and then the following message:

We will attempt to step through and interpret the code.

It is still possible that Axiom can evaluate the loop but in *interpret-code mode*. See section 6.10 on page 247 where this is discussed in terms of compiling versus interpreting functions.

5.4.2   return in Loops

A `return` expression is used to exit a function with a particular value. In particular, if a `return` is in a loop within the function, the loop is terminated whenever the `return` is evaluated.

Suppose we start with this.

```
f() ==
  i := 1
  repeat
    if factorial(i) > 1000 then return i
    i := i + 1
```

                                                                    Type: Void

When `factorial(i)` is big enough, control passes from inside the loop all the way outside the function, returning the value of i (or so we think).

```
f()
```

                                                                    Type: Void

What went wrong? Isn't it obvious that this function should return an integer? Well, Axiom makes no attempt to analyze the structure of a loop to determine if it always returns a value because, in general, this is impossible. So Axiom has this simple rule: the type of the function is determined by the type of its body, in this case a block. The normal value of a block is the value of its last expression, in this case, a loop. And the value of every loop is the unique value of `Void`.! So the return type of `f` is `Void`.

There are two ways to fix this. The best way is for you to tell Axiom what the return type of f is. You do this by giving f a declaration `f:() -> Integer` prior to calling for its value. This tells Axiom: "trust me—an integer is returned." We'll explain more about this in the next chapter. Another clumsy way is to add a dummy expression as follows.

Since we want an integer, let's stick in a dummy final expression that is an integer and will never be evaluated.

```
f() ==
  i := 1
  repeat
    if factorial(i) > 1000 then return i
    i := i + 1
  0
```

```
                                                        Type:  Void
```

When we try **f** again we get what we wanted. See 6.15 on page 265 for more information.

```
f()
```

```
   Compiling function f with type () -> NonNegativeInteger
```

$$7$$

```
                                              Type:  PositiveInteger
```

5.4.3 break in Loops

The `break` keyword is often more useful in terminating a loop. A `break` causes control to transfer to the expression immediately following the loop. As loops always return the unique value of `Void.`, you cannot return a value with `break`. That is, `break` takes no argument.

This example is a modification of the last example in the previous section 5.4.2 on page 206. Instead of using `return`, we'll use `break`.

```
f() ==
  i := 1
  repeat
    if factorial(i) > 1000 then break
    i := i + 1
  i
```

```
   Compiled code for f has been cleared.
   1 old definition(s) deleted for function or rule f
```

```
                                                        Type:  Void
```

The loop terminates when `factorial(i)` gets big enough, the last line of the function evaluates to the corresponding "good" value of i, and the function terminates, returning that value.

```
f()
```

```
   Compiling function f with type () -> PositiveInteger
```

```
+++ |*0;f;1;G82322| redefined
```

$$7$$

```
                                         Type:   PositiveInteger
```

You can only use `break` to terminate the evaluation of one loop. Let's consider a loop within a loop, that is, a loop with a nested loop. First, we initialize two counter variables.

```
(i,j) := (1, 1)
```

$$1$$

```
                                         Type:   PositiveInteger
```

Nested loops must have multiple `break` expressions at the appropriate nesting level. How would you rewrite this so `(i + j) > 10` is only evaluated once?

```
repeat
  repeat
    if (i + j) > 10 then break
    j := j + 1
  if (i + j) > 10 then break
  i := i + 1
```

```
                                                Type:   Void
```

5.4.4   break vs. => in Loop Bodies

Compare the following two loops:

```
i := 1                          i := 1
repeat                          repeat
  i := i + 1                      i := i + 1
  i > 3 => i                      if i > 3 then break
  output(i)                       output(i)
```

In the example on the left, the values 2 and 3 for i are displayed but then the "=>" does not allow control to reach the call to **output** again. The loop will not terminate until you run out of space or interrupt the execution. The variable i

will continue to be incremented because the "=>" only means to leave the *block*, not the loop.

In the example on the right, upon reaching 4, the `break` will be executed, and both the block and the loop will terminate. This is one of the reasons why both "=>" and `break` are provided. Using a `while` clause (see below) with the "=>" lets you simulate the action of `break`.

5.4.5 More Examples of break

Here we give four examples of `repeat` loops that terminate when a value exceeds a given bound.

First, initialize i as the loop counter.

```
i := 0
```

                                                0

                                                        Type: NonNegativeInteger

Here is the first loop. When the square of i exceeds 100, the loop terminates.

```
repeat
  i := i + 1
  if i**2 > 100 then break
```

                                                                Type: Void

Upon completion, i should have the value 11.

```
i
```

                                                11

                                                        Type: NonNegativeInteger

Do the same thing except use "=>" instead an `if-then` expression.

```
i := 0
```

                                                0

                                                        Type: NonNegativeInteger

```
repeat
  i := i + 1
  i**2 > 100 => break
```

                                                    Type: Void

i

                                    11

                                        Type: NonNegativeInteger

As a third example, we use a simple loop to compute $n!$.

```
(n, i, f) := (100, 1, 1)
```

                                    1

                                        Type: PositiveInteger

Use i as the iteration variable and f to compute the factorial.

```
repeat
  if i > n then break
  f := f * i
  i := i + 1
```

                                                    Type: Void

Look at the value of f.

f


```
  933262154439441526816992388562667004907159682643816214 68_
  592963895217599993229915608941463976156518286253697920 82_
  7223758251185210916864000000000000000000000000000000
```

                                        Type: PositiveInteger

Finally, we show an example of nested loops. First define a four by four matrix.

```
m := matrix [ [21,37,53,14], [8,-24,22,-16], [2,10,15,14],
[26,33,55,-13] ]
```

$$\begin{bmatrix} 21 & 37 & 53 & 14 \\ 8 & -24 & 22 & -16 \\ 2 & 10 & 15 & 14 \\ 26 & 33 & 55 & -13 \end{bmatrix}$$

                                                              Type: Matrix Integer

Next, set row counter r and column counter c to 1. Note: if we were writing a function, these would all be local variables rather than global workspace variables.

```
(r, c) := (1, 1)
```

                                        1

                                                              Type: PositiveInteger

Also, let `lastrow` and `lastcol` be the final row and column index.

```
(lastrow, lastcol) := (nrows(m), ncols(m))
```

                                        4

                                                              Type: PositiveInteger

Scan the rows looking for the first negative element. We remark that you can reformulate this example in a better, more concise form by using a `for` clause with `repeat`. See 5.4.8 on page 215 for more information.

```
repeat
  if r > lastrow then break
  c := 1
  repeat
    if c > lastcol then break
    if elt(m,r,c) < 0 then
      output [r, c, elt(m,r,c)]
      r := lastrow
      break      -- don't look any further
    c := c + 1
  r := r + 1

  [2,2,- 24]
```

                                                              Type: Void

5.4.6   iterate in Loops

Axiom provides an `iterate` expression that skips over the remainder of a loop body and starts the next loop iteration.

We first initialize a counter.

```
i := 0
```

$$0$$

<p align="right">Type: NonNegativeInteger</p>

Display the even integers from 2 to 5.

```
repeat
  i := i + 1
  if i > 5 then break
  if odd?(i) then iterate
  output(i)
```

```
  2
  4
```

<p align="right">Type: Void</p>

5.4.7   while Loops

The `repeat` in a loop can be modified by adding one or more `while` clauses. Each clause contains a *predicate* immediately following the `while` keyword. The predicate is tested *before* the evaluation of the body of the loop. The loop body is evaluated whenever the predicates in a `while` clause are all `true`.

The syntax for a simple loop using `while` is

<p align="center">`while` predicate `repeat` loopBody</p>

The *predicate* is evaluated before *loopBody* is evaluated. A `while` loop terminates immediately when *predicate* evaluates to `false` or when a `break` or `return` expression is evaluated in *loopBody*. The value returned by the loop is the unique value of `Void`.

Here is a simple example of using `while` in a loop. We first initialize the counter.

```
i := 1
```

$$1$$

The steps involved in computing this example are
(1) set i to 1,
(2) test the condition $i < 1$ and determine that it is not **true**, and
(3) do not evaluate the loop body and therefore do not display "*hello*".

```
while i < 1 repeat
  output "hello"
  i := i + 1
```

                                        Type: Void

If you have multiple predicates to be tested use the logical **and** operation to
separate them. Axiom evaluates these predicates from left to right.

```
(x, y) := (1, 1)
```

$$1$$

                                        Type: PositiveInteger

```
while x < 4 and y < 10 repeat
  output [x,y]
  x := x + 1
  y := y + 2
```

```
    [1,1]
    [2,3]
    [3,5]
```

                                        Type: Void

A **break** expression can be included in a loop body to terminate a loop even if
the predicate in any **while** clauses are not **false**.

```
(x, y) := (1, 1)
```

$$1$$

<div align="right">Type: PositiveInteger</div>

This loop has multiple `while` clauses and the loop terminates before any one of their conditions evaluates to `false`.

```
while x < 4 while y < 10 repeat
  if x + y > 7 then break
  output [x,y]
  x := x + 1
  y := y + 2

  [1,1]
  [2,3]
```

<div align="right">Type: Void</div>

Here's a different version of the nested loops that looked for the first negative element in a matrix.

```
m := matrix [ [21,37,53,14], [8,-24,22,-16], [2,10,15,14],
[26,33,55,-13] ]
```

$$\begin{bmatrix} 21 & 37 & 53 & 14 \\ 8 & -24 & 22 & -16 \\ 2 & 10 & 15 & 14 \\ 26 & 33 & 55 & -13 \end{bmatrix}$$

<div align="right">Type: Matrix Integer</div>

Initialized the row index to 1 and get the number of rows and columns. If we were writing a function, these would all be local variables.

```
r := 1
```

$$1$$

<div align="right">Type: PositiveInteger</div>

```
(lastrow, lastcol) := (nrows(m), ncols(m))
```

$$4$$

<div align="right">Type: PositiveInteger</div>

Scan the rows looking for the first negative element.

```
while r <= lastrow repeat
  c := 1  -- index of first column
  while c <= lastcol repeat
    if elt(m,r,c) < 0 then
      output [r, c, elt(m,r,c)]
      r := lastrow
      break      -- don't look any further
    c := c + 1
  r := r + 1

  [2,2,- 24]
```

                                                        Type: Void

5.4.8   for Loops

Axiom provides the `for` and `in` keywords in `repeat` loops, allowing you to
iterate across all elements of a list, or to have a variable take on integral values
from a lower bound to an upper bound. We shall refer to these modifying
clauses of `repeat` loops as `for` clauses. These clauses can be present in addition
to `while` clauses. As with all other types of `repeat` loops, `break` can be used
to prematurely terminate the evaluation of the loop.

The syntax for a simple loop using `for` is

for *iterator* `repeat` *loopBody*

The *iterator* has several forms. Each form has an end test which is evaluated
before *loopBody* is evaluated. A `for` loop terminates immediately when the
end test succeeds (evaluates to `true`) or when a `break` or `return` expression
is evaluated in *loopBody*. The value returned by the loop is the unique value
of `Void`.

5.4.9   for i in n..m repeat

If `for` is followed by a variable name, the `in` keyword and then an integer
segment of the form $n..m$, the end test for this loop is the predicate $i > m$. The
body of the loop is evaluated $m - n + 1$ times if this number is greater than 0.
If this number is less than or equal to 0, the loop body is not evaluated at all.

The variable i has the value $n, n + 1, ..., m$ for successive iterations of the loop
body.The loop variable is a *local variable* within the loop body: its value is not

available outside the loop body and its value and type within the loop body completely mask any outer definition of a variable with the same name.

This loop prints the values of 10^3, 11^3, and 12^3:

```
for i in 10..12 repeat output(i**3)
```

```
    1000
    1331
    1728
```

<div align="right">Type: Void</div>

Here is a sample list.

```
a := [1,2,3]
```

$$[1, 2, 3]$$

<div align="right">Type: List PositiveInteger</div>

Iterate across this list, using "." to access the elements of a list and the "#" operation to count its elements.

```
for i in 1..#a repeat output(a.i)
```

```
    1
    2
    3
```

<div align="right">Type: Void</div>

This type of iteration is applicable to anything that uses ".". You can also use it with functions that use indices to extract elements.

Define m to be a matrix.

```
m := matrix [ [1,2],[4,3],[9,0] ]
```

$$\begin{bmatrix} 1 & 2 \\ 4 & 3 \\ 9 & 0 \end{bmatrix}$$

<div align="right">Type: Matrix Integer</div>

Display the rows of m.

```
for i in 1..nrows(m) repeat output row(m,i)
```

```
   [1,2]
   [4,3]
   [9,0]
```

<div align="right">Type: Void</div>

You can use `iterate` with `for`-loops.

Display the even integers in a segment.

```
for i in 1..5 repeat
  if odd?(i) then iterate
  output(i)
```

```
   2
   4
```

<div align="right">Type: Void</div>

See section 9.69 on page 783 for more information about segments.

5.4.10   for i in n..m by s repeat

By default, the difference between values taken on by a variable in loops such as `for i in n..m repeat ...` is 1. It is possible to supply another, possibly negative, step value by using the `by` keyword along with `for` and `in` . Like the upper and lower bounds, the step value following the `by` keyword must be an integer. Note that the loop `for i in 1..2 by 0 repeat output(i)` will not terminate by itself, as the step value does not change the index from its initial value of 1.

This expression displays the odd integers between two bounds.

```
for i in 1..5 by 2 repeat output(i)
```

```
   1
   3
   5
```

<div align="right">Type: Void</div>

Use this to display the numbers in reverse order.

```
for i in 5..1 by -2 repeat output(i)
```

```
   5
   3
   1
```

<div align="right">Type: Void</div>

5.4.11   for i in n.. repeat

If the value after the "`..`" is omitted, the loop has no end test. A potentially infinite loop is thus created. The variable is given the successive values $n, n+1, n+2, ...$ and the loop is terminated only if a **break** or **return** expression is evaluated in the loop body. However you may also add some other modifying clause on the **repeat** (for example, a **while** clause) to stop the loop.

This loop displays the integers greater than or equal to 15 and less than the first prime greater than 15.

```
for i in 15..  while not prime?(i) repeat output(i)
```

```
   15
   16
```

<div align="right">Type: Void</div>

5.4.12   for x in l repeat

Another variant of the **for** loop has the form:

$$\text{\textbf{for} } x \text{ \textbf{in}}\quad list \text{ \textbf{repeat} } loopBody$$

This form is used when you want to iterate directly over the elements of a list. In this form of the **for** loop, the variable x takes on the value of each successive element in l. The end test is most simply stated in English: "are there no more x in l?"

If l is this list,

```
l := [0,-5,3]
```

$$[0, -5, 3]$$

                                                        Type: List Integer

display all elements of `l`, one per line.

```
for x in l repeat output(x)
```

```
   0
   - 5
   3
```

                                                             Type: Void

Since the list constructing expression **expand**`[n..m]` creates the list $[n, n+1, ..., m]$. Note that this list is empty if $n > m$. You might be tempted to think that the loops

```
for i in n..m repeat output(i)
```

and

```
for x in expand [n..m] repeat output(x)
```

are equivalent. The second form first creates the list **expand**`[n..m]` (no matter how large it might be) and then does the iteration. The first form potentially runs in much less space, as the index variable i is simply incremented once per loop and the list is not actually created. Using the first form is much more efficient.

Of course, sometimes you really want to iterate across a specific list. This displays each of the factors of 2400000.

```
for f in factors(factor(2400000)) repeat output(f)
```

```
   [factor= 2,exponent= 8]
   [factor= 3,exponent= 1]
   [factor= 5,exponent= 5]
```

                                                             Type: Void

5.4.13 "Such that" Predicates

A `for` loop can be followed by a "|" and then a predicate. The predicate qualifies the use of the values from the iterator following the `for`. Think of the vertical bar "|" as the phrase "such that."

This loop expression prints out the integers n in the given segment such that n is odd.

```
for n in 0..4 | odd?  n repeat output n
```

```
  1
  3
```

<div align="right">Type: Void</div>

A `for` loop can also be written

$$\text{for } iterator \mid predicate \text{ repeat } loopBody$$

which is equivalent to:

$$\text{for } iterator \text{ repeat if } predicate \text{ then } loopBody \text{ else } iterate$$

The predicate need not refer only to the variable in the `for` clause: any variable in an outer scope can be part of the predicate.

In this example, the predicate on the inner `for` loop uses i from the outer loop and the j from the `for` clause that it directly modifies.

```
for i in 1..50 repeat
  for j in 1..50 | factorial(i+j) < 25 repeat
    output [i,j]
```

```
  [1,1]
  [1,2]
  [1,3]
  [2,1]
  [2,2]
  [3,1]
```

<div align="right">Type: Void</div>

5.4.14 Parallel Iteration

The last example of the previous section 5.4.13 on page 220 gives an example of *nested iteration*: a loop is contained in another loop. Sometimes you want to iterate across two lists in parallel, or perhaps you want to traverse a list while incrementing a variable.

The general syntax of a repeat loop is

$$iterator_1 \ iterator_2 \ \ldots \ iterator_N \ repeat \ loopBody$$

where each *iterator* is either a `for` or a `while` clause. The loop terminates immediately when the end test of any *iterator* succeeds or when a `break` or `return` expression is evaluated in *loopBody*. The value returned by the loop is the unique value of `Void`.

Here we write a loop to iterate across two lists, computing the sum of the pairwise product of elements. Here is the first list.

```
l := [1,3,5,7]
```

$$[1, 3, 5, 7]$$

                                        Type: List PositiveInteger

And the second.

```
m := [100,200]
```

$$[100, 200]$$

                                        Type: List PositiveInteger

The initial value of the sum counter.

```
sum := 0
```

$$0$$

                                        Type: NonNegativeInteger

The last two elements of l are not used in the calculation because m has two fewer elements than l.

```
for x in l for y in m repeat
    sum := sum + x*y
```

                                                    Type: Void

Display the "dot product."

```
sum
```

                            700

                                        Type: NonNegativeInteger

Next, we write a loop to compute the sum of the products of the loop elements
with their positions in the loop.

```
l := [2,3,5,7,11,13,17,19,23,29,31,37]
```

$$[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]$$

                                        Type: List PositiveInteger

The initial sum.

```
sum := 0
```

                            0

                                        Type: NonNegativeInteger

Here looping stops when the list l is exhausted, even though the $for i in 0..$ spec-
ifies no terminating condition.

```
for i in 0..  for x in l repeat sum := i * x
```

                                                    Type: Void

Display this weighted sum.

```
sum
```

```
                                    Type:  NonNegativeInteger
```

When "|" is used to qualify any of the `for` clauses in a parallel iteration, the
variables in the predicates can be from an outer scope or from a `for` clause in
or to the left of a modified clause.

This is correct:

```
for i in 1..10 repeat
  for j in 200..300 | odd? (i+j) repeat
    output [i,j]
```

This is not correct since the variable j has not been defined outside the inner
loop.

```
for i in 1..10 | odd? (i+j) repeat  -- wrong, j not defined
  for j in 200..300 repeat
    output [i,j]
```

5.4.15 Mixing Loop Modifiers

This example shows that it is possible to mix several of the forms of `repeat`
modifying clauses on a loop.

```
for i in 1..10
    for j in 151..160 | odd? j
      while i + j < 160 repeat
        output [i,j]

   [1,151]
   [3,153]
```

```
                                            Type:  Void
```

Here are useful rules for composing loop expressions:

1. `while` predicates can only refer to variables that are global (or in an outer
   scope) or that are defined in `for` clauses to the left of the predicate.

2. A "such that" predicate (something following "|") must directly follow a
   `for` clause and can only refer to variables that are global (or in an outer
   scope) or defined in the modified `for` clause or any `for` clause to the left.

5.5   Creating Lists and Streams with Iterators

All of what we did for loops in 5.4 on page 205 can be transformed into expressions that create lists and streams. The `repeat`, `break` or `iterate` words are not used but all the other ideas carry over. Before we give you the general rule, here are some examples which give you the idea.

This creates a simple list of the integers from 1 to 10.

```
list := [i for i in 1..10]
```

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

<div align="right">Type: List PositiveInteger</div>

Create a stream of the integers greater than or equal to 1.

```
stream := [i for i in 1..]
```

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \ldots]$$

<div align="right">Type: Stream PositiveInteger</div>

This is a list of the prime integers between 1 and 10, inclusive.

```
[i for i in 1..10 | prime?  i]
```

$$[2, 3, 5, 7]$$

<div align="right">Type: List PositiveInteger</div>

This is a stream of the prime integers greater than or equal to 1.

```
[i for i in 1..  | prime?  i]
```

$$[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, \ldots]$$

<div align="right">Type: Stream PositiveInteger</div>

This is a list of the integers between 1 and 10, inclusive, whose squares are less than 700.

```
[i for i in 1..10 while i*i < 700]
```

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Type: List PositiveInteger

This is a stream of the integers greater than or equal to 1 whose squares are less than 700.

```
[i for i in 1..  while i*i < 700]
```

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \ldots]$$

Type: Stream PositiveInteger

Here is the general rule.

The general syntax of a collection is

[*collectExpression iterator$_1$ iterator$_2$ ...iterator$_N$*]

where each *iterator$_i$* is either a `for` or a `while` clause. The loop terminates immediately when the end test of any *iterator$_i$* succeeds or when a `return` expression is evaluated in *collectExpression*. The value returned by the collection is either a list or a stream of elements, one for each iteration of the *collectExpression*.

Be careful when you use `while` to create a stream. By default, Axiom tries to compute and display the first ten elements of a stream. If the `while` condition is not satisfied quickly, Axiom can spend a long (possibly infinite) time trying to compute the elements. Use `)set streams calculate` to change the default to something else. This also affects the number of terms computed and displayed for power series. For the purposes of this book, we have used this system command to display fewer than ten terms.

Use nested iterators to create lists of lists which can then be given as an argument to **matrix**.

```
matrix [ [x**i+j for i in 1..3] for j in 10..12]
```

$$\begin{bmatrix} x+10 & x^2+10 & x^3+10 \\ x+11 & x^2+11 & x^3+11 \\ x+12 & x^2+12 & x^3+12 \end{bmatrix}$$

Type: Matrix Polynomial Integer

You can also create lists of streams, streams of lists and streams of streams. Here is a stream of streams.

```
[ [i/j for i in j+1..]  for j in 1..]
```

$$\left[[2,3,4,5,6,7,8,9,10,11,\ldots], \left[\tfrac{3}{2},2,\tfrac{5}{2},3,\tfrac{7}{2},4,\tfrac{9}{2},5,\tfrac{11}{2},6,\ldots\right],\right.$$

$$\left[\frac{4}{3},\frac{5}{3},2,\frac{7}{3},\frac{8}{3},3,\frac{10}{3},\frac{11}{3},4,\frac{13}{3},\ldots\right],\left[\frac{5}{4},\frac{3}{2},\frac{7}{4},2,\frac{9}{4},\frac{5}{2},\frac{11}{4},3,\frac{13}{4},\frac{7}{2},\ldots\right],$$

$$\left[\frac{6}{5},\frac{7}{5},\frac{8}{5},\frac{9}{5},2,\frac{11}{5},\frac{12}{5},\frac{13}{5},\frac{14}{5},3,\ldots\right],\left[\frac{7}{6},\frac{4}{3},\frac{3}{2},\frac{5}{3},\frac{11}{6},2,\frac{13}{6},\frac{7}{3},\frac{5}{2},\frac{8}{3},\ldots\right],$$

$$\left[\frac{8}{7},\frac{9}{7},\frac{10}{7},\frac{11}{7},\frac{12}{7},\frac{13}{7},2,\frac{15}{7},\frac{16}{7},\frac{17}{7},\ldots\right],\left[\frac{9}{8},\frac{5}{4},\frac{11}{8},\frac{3}{2},\frac{13}{8},\frac{7}{4},\frac{15}{8},2,\frac{17}{8},\frac{9}{4},\ldots\right],$$

$$\left[\frac{10}{9},\frac{11}{9},\frac{4}{3},\frac{13}{9},\frac{14}{9},\frac{5}{3},\frac{16}{9},\frac{17}{9},2,\frac{19}{9},\ldots\right],$$

$$\left.\left[\frac{11}{10},\frac{6}{5},\frac{13}{10},\frac{7}{5},\frac{3}{2},\frac{8}{5},\frac{17}{10},\frac{9}{5},\frac{19}{10},2,\ldots\right],\ldots\right]$$

```
                              Type:  Stream Stream Fraction Integer
```

You can use parallel iteration across lists and streams to create new lists.

```
[i/j for i in 3..  by 10 for j in 2..]
```

$$\left[\frac{3}{2},\frac{13}{3},\frac{23}{4},\frac{33}{5},\frac{43}{6},\frac{53}{7},\frac{63}{8},\frac{73}{9},\frac{83}{10},\frac{93}{11},\ldots\right]$$

```
                                   Type:  Stream Fraction Integer
```

Iteration stops if the end of a list or stream is reached.

```
[i**j for i in 1..7 for j in 2..  ]
```

$$[1,8,81,1024,15625,279936,5764801]$$

```
                                          Type:  Stream Integer
```

As with loops, you can combine these modifiers to make very complicated conditions.

```
[ [ [i,j] for i in 10..15 | prime?  i] for j in 17..22 | j =
squareFreePart j]
```

$$[[[11,17],[13,17]],[[11,19],[13,19]],[[11,21],[13,21]],[[11,22],[13,22]]]$$

```
                                  Type:  List List List PositiveInteger
```

See List (section 9.47 on page 675) and Stream (section 9.76 on page 801) for more information on creating and manipulating lists and streams, respectively.

5.6 An Example: Streams of Primes

We conclude this chapter with an example of the creation and manipulation of infinite streams of prime integers. This might be useful for experiments with numbers or other applications where you are using sequences of primes over and over again. As for all streams, the stream of primes is only computed as far out as you need. Once computed, however, all the primes up to that point are saved for future reference.

Two useful operations provided by the Axiom library are **prime?** and **nextPrime**. A straight-forward way to create a stream of prime numbers is to start with the stream of positive integers [2, ..] and filter out those that are prime.

Create a stream of primes.

```
primes :  Stream Integer := [i for i in 2..  | prime?  i]
```

$$[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, \ldots]$$

Type: Stream Integer

A more elegant way, however, is to use the **generate** operation from `Stream`. Given an initial value a and a function f, **generate** constructs the stream $[a, f(a), f(f(a)), ...]$. This function gives you the quickest method of getting the stream of primes.

This is how you use **generate** to generate an infinite stream of primes.

```
primes := generate(nextPrime,2)
```

$$[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, \ldots]$$

Type: Stream Integer

Once the stream is generated, you might only be interested in primes starting at a particular value.

```
smallPrimes := [p for p in primes | p > 1000]
```

$$[1009, 1013, 1019, 1021, 1031, 1033, 1039, 1049, 1051, 1061, \ldots]$$

Type: Stream Integer

Here are the first 11 primes greater than 1000.

```
[p for p in smallPrimes for i in 1..11]
```

$$[1009, 1013, 1019, 1021, 1031, 1033, 1039, 1049, 1051, 1061, \ldots]$$

<div align="right">Type: Stream Integer</div>

Here is a stream of primes between 1000 and 1200.

```
[p for p in smallPrimes while p < 1200]
```

$$[1009, 1013, 1019, 1021, 1031, 1033, 1039, 1049, 1051, 1061, \ldots]$$

<div align="right">Type: Stream Integer</div>

To get these expanded into a finite stream, you call **complete** on the stream.

```
complete %
```

$$[1009, 1013, 1019, 1021, 1031, 1033, 1039, 1049, 1051, 1061, \ldots]$$

<div align="right">Type: Stream Integer</div>

Twin primes are consecutive odd number pairs which are prime. Here is the stream of twin primes.

```
twinPrimes := [ [p,p+2] for p in primes | prime?(p + 2)]
```

$$[[3, 5], [5, 7], [11, 13], [17, 19], [29, 31], [41, 43], [59, 61], [71, 73],$$

$$[101, 103], [107, 109], \ldots]$$

<div align="right">Type: Stream List Integer</div>

Since we already have the primes computed we can avoid the call to **prime?** by using a double iteration. This time we'll just generate a stream of the first of the twin primes.

```
firstOfTwins:= [p for p in primes for q in rest primes | q=p+2]
```

$$[3, 5, 11, 17, 29, 41, 59, 71, 101, 107, \ldots]$$

<div align="right">Type: Stream Integer</div>

Let's try to compute the infinite stream of triplet primes, the set of primes p such that $[p, p + 2, p + 4]$ are primes. For example, $[3, 5, 7]$ is a triple prime. We could do this by a triple **for** iteration. A more economical way is to use **firstOfTwins**. This time however, put a semicolon at the end of the line.

Create the stream of firstTriplets. Put a semicolon at the end so that no elements are computed.

```
firstTriplets := [p for p in firstOfTwins for q in rest
firstOfTwins | q = p+2];
```

                                        Type: Stream Integer

What happened? As you know, by default Axiom displays the first ten elements
of a stream when you first display it. And, therefore, it needs to compute
them! If you want *no* elements computed, just terminate the expression by a
semicolon (";"). The semi-colon prevents the display of the result of evaluating
the expression. Since no stream elements are needed for display (or anything
else, so far), none are computed.

Compute the first triplet prime.

```
firstTriplets.1
```

                            3

                                        Type: PositiveInteger

If you want to compute another, just ask for it. But wait a second! Given three
consecutive odd integers, one of them must be divisible by 3. Thus there is only
one triplet prime. But suppose that you did not know this and wanted to know
what was the tenth triplet prime.

```
firstTriples.10
```

To compute the tenth triplet prime, Axiom first must compute the second, the
third, and so on. But since there isn't even a second triplet prime, Axiom will
compute forever. Nonetheless, this effort can produce a useful result. After
waiting a bit, hit **Ctrl-c**. The system responds as follows.

```
   >> System error:
   Console interrupt.
   You are being returned to the top level of
   the interpreter.
```

If you want to know how many primes have been computed, type:

```
numberOfComputedEntries primes
```

and, for this discussion, let's say that the result is 2045. How big is the 2045-th
prime?

```
primes.2045
```

17837

```
                                   Type:  PositiveInteger
```

What you have learned is that there are no triplet primes between 5 and 17837. Although this result is well known (some might even say trivial), there are many experiments you could make where the result is not known. What you see here is a paradigm for testing of hypotheses. Here our hypothesis could have been: "there is more than one triplet prime." We have tested this hypothesis for 17837 cases. With streams, you can let your machine run, interrupt it to see how far it has progressed, then start it up and let it continue from where it left off.

Chapter 6

User-Defined Functions, Macros and Rules

In this chapter we show you how to write functions and macros, and we explain how Axiom looks for and applies them. We show some simple one-line examples of functions, together with larger ones that are defined piece-by-piece or through the use of piles.

6.1 Functions vs. Macros

A function is a program to perform some computation. Most functions have names so that it is easy to refer to them. A simple example of a function is one named **abs** which computes the absolute value of an integer.

This is a use of the "absolute value" library function for integers.

```
abs(-8)
```

$$8$$

                                        Type: PositiveInteger

This is an unnamed function that does the same thing, using the "maps-to" syntax `+->` that we discuss in section .

```
(x +-> if x < 0 then -x else x)(-8)
```

$$8$$

                                        Type: PositiveInteger

Functions can be used alone or serve as the building blocks for larger programs. Usually they return a value that you might want to use in the next stage of a computation, but not always (for example, see 9.20 on page 539 and 9.86 on page 839). They may also read data from your keyboard, move information from one place to another, or format and display results on your screen.

In Axiom, as in mathematics, functions are usually parameterized. Each time you *call* (some people say *apply* or invoke) a function, you give values to the parameters (variables). Such a value is called an *argument* of the function. Axiom uses the arguments for the computation. In this way you get different results depending on what you "feed" the function.

Functions can have local variables or refer to global variables in the workspace. Axiom can often compile functions so that they execute very efficiently. Functions can be passed as arguments to other functions.

Macros are textual substitutions. They are used to clarify the meaning of constants or expressions and to be templates for frequently used expressions. Macros can be parameterized but they are not objects that can be passed as arguments to functions. In effect, macros are extensions to the Axiom expression parser.

6.2 Macros

A *macro* provides general textual substitution of an Axiom expression for a name. You can think of a macro as being a generalized abbreviation. You can only have one macro in your workspace with a given name, no matter how many arguments it has.

The two general forms for macros are

$$\texttt{macro}\ name\ \texttt{==}\ body$$
$$\texttt{macro}\ name(arg1,...)\ \texttt{==}\ body$$

where the body of the macro can be any Axiom expression.

For example, suppose you decided that you like to use `df` for `D`. You define the macro `df` like this.

```
macro df == D
```

                                                              Type: Void

Whenever you type `df`, the system expands it to `D`.

```
df(x**2 + x + 1,x)
```

$$2\ x + 1$$

<div align="right">Type: Polynomial Integer</div>

Macros can be parameterized and so can be used for many different kinds of objects.

```
macro ff(x) == x**2 + 1
```

<div align="right">Type: Void</div>

Apply it to a number, a symbol, or an expression.

```
ff z
```

$$z^2 + 1$$

<div align="right">Type: Polynomial Integer</div>

Macros can also be nested, but you get an error message if you run out of space because of an infinite nesting loop.

```
macro gg(x) == ff(2*x - 2/3)
```

<div align="right">Type: Void</div>

This new macro is fine as it does not produce a loop.

```
gg(1/w)
```

$$\frac{13\ w^2 - 24\ w + 36}{9\ w^2}$$

<div align="right">Type: Fraction Polynomial Integer</div>

This, however, loops since `gg` is defined in terms of `ff`.

```
macro ff(x) == gg(-x)
```

<div align="right">Type: Void</div>

The body of a macro can be a block.

```
macro next == (past := present; present := future; future := past
+ present)
```

<div align="right">Type: Void</div>

Before entering `next`, we need values for `present` and `future`.

```
present :  Integer := 0
```

<div align="center">0</div>

<div align="right">Type: Integer</div>

```
future :  Integer := 1
```

<div align="center">1</div>

<div align="right">Type: Integer</div>

Repeatedly evaluating `next` produces the next Fibonacci number.

```
next
```

<div align="center">1</div>

<div align="right">Type: Integer</div>

And the next one.

```
next
```

<div align="center">2</div>

<div align="right">Type: Integer</div>

Here is the infinite stream of the rest of the Fibonacci numbers.

```
[next for i in 1..]
```

$$[3, 5, 8, 13, 21, 34, 55, 89, 144, 233, \ldots]$$

<div align="right">Type: Stream Integer</div>

Bundle all the above lines into a single macro.

```
macro fibStream ==
  present : Integer := 1
  future : Integer := 1
  [next for i in 1..] where
    macro next ==
      past := present
      present := future
      future := past + present
```

<div align="right">Type: Void</div>

Use **concat** to start with the first two Fibonacci numbers.

```
concat([0,1],fibStream)
```

$$[0, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots]$$

<div align="right">Type: Stream Integer</div>

The library operation **fibonacci** is an easier way to compute these numbers.

```
[fibonacci i for i in 1..]
```

$$[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots]$$

<div align="right">Type: Stream Integer</div>

6.3 Introduction to Functions

Each name in your workspace can refer to a single object. This may be any kind of object including a function. You can use interactively any function from the library or any that you define in the workspace. In the library the same name can have very many functions, but you can have only one function with a given name, although it can have any number of arguments that you choose.

If you define a function in the workspace that has the same name and number of arguments as one in the library, then your definition takes precedence. In fact, to get the library function you must *package-call* it (see section 2.9 on page 162).

To use a function in Axiom, you apply it to its arguments. Most functions are applied by entering the name of the function followed by its argument or arguments.

```
factor(12)
```

$$2^2\ 3$$

<div align="right">Type: Factored Integer</div>

Some functions like "+" have *infix operators* as names.

```
3 + 4
```

$$7$$

<div align="right">Type: PositiveInteger</div>

The function "+" has two arguments. When you give it more than two arguments, Axiom groups the arguments to the left. This expression is equivalent to $(1 + 2) + 7$.

```
1 + 2 + 7
```

$$10$$

<div align="right">Type: PositiveInteger</div>

All operations, including infix operators, can be written in prefix form, that is, with the operation name followed by the arguments in parentheses. For example, $2 + 3$ can alternatively be written as $+(2, 3)$. But $+(2, 3, 4)$ is an error since + takes only two arguments.

Prefix operations are generally applied before the infix operation. Thus the form **factorial** $3 + 1$ means **factorial**$(3) + 1$ producing 7, and $-2 + 5$ means $(-2) + 5$ producing 3. An example of a prefix operator is prefix "-". For example, $-2 + 5$ converts to $(-2) + 5$ producing the value 3. Any prefix function taking two arguments can be written in an infix manner by putting an ampersand "&" before the name. Thus $\mathtt{D}(2 * x, x)$ can be written as $2 * x\ \&\mathtt{D}x$ returning 2.

Every function in Axiom is identified by a *name* and *type*. (An exception is an "anonymous function" discussed in 6.17 on page 275.) The type of a function is always a mapping of the form Source → Target where Source and Target are types. To enter a type from the keyboard, enter the arrow by using a hyphen "-" followed by a greater-than sign ">", e.g. Integer -> Integer.

Let's go back to "+". There are many "+" functions in the Axiom library: one for integers, one for floats, another for rational numbers, and so on. These "+" functions have different types and thus are different functions. You've seen examples of this *overloading* before—using the same name for different functions. Overloading is the rule rather than the exception. You can add two integers, two polynomials, two matrices or two power series. These are all done with the same function name but with different functions.

6.4 Declaring the Type of Functions

In 2.3 on page 142 we discussed how to declare a variable to restrict the kind of values that can be assigned to it. In this section we show how to declare a variable that refers to function objects.

A function is an object of type

$$\mathsf{Source} \to \mathsf{Type}$$

where `Source` and `Target` can be any type. A common type for `Source` is `Tuple`$(T_1, \ldots, T_n)$, usually written $(T_1, \ldots, T_n)$, to indicate a function of n arguments.

If g takes an `Integer`, a `Float` and another `Integer`, and returns a `String`, the declaration is written:

```
g:  (Integer,Float,Integer) -> String
```

```
                                        Type:  Void
```

The types need not be written fully; using abbreviations, the above declaration is:

```
g:  (INT,FLOAT,INT) -> STRING
```

```
                                        Type:  Void
```

It is possible for a function to take no arguments. If h takes no arguments but returns a `Polynomial Integer`, any of the following declarations is acceptable.

```
h:  () -> POLY INT
```

```
                                        Type:  Void
```

```
h:  () -> Polynomial INT
```

```
                                        Type:  Void
```

```
h:  () -> POLY Integer
```

```
                                            Type:  Void
```

> Functions can also be declared when they are being defined. The syntax for combined declaration/definition is:
>
> $$functionName(parm_1:\ parmType_1,\ \ldots,\ parm_N:\ parmType_N):$$
> $$functionReturnType$$

The following definition fragments show how this can be done for the functions g and h above.

```
g(arg1: INT, arg2: FLOAT, arg3: INT): STRING == ...

h(): POLY INT == ...
```

A current restriction on function declarations is that they must involve fully specified types (that is, cannot include modes involving explicit or implicit "?"). For more information on declaring things in general, see .

6.5 One-Line Functions

As you use Axiom, you will find that you will write many short functions to codify sequences of operations that you often perform. In this section we write some simple one-line functions.

This is a simple recursive factorial function for positive integers.

```
fac n == if n < 3 then n else n * fac(n-1)
```

```
                                            Type:  Void
```

```
fac 10
```

$$3628800$$

```
                                    Type:  PositiveInteger
```

This function computes $1 + 1/2 + 1/3 + \ldots + 1/n$.

```
s n == reduce(+,[1/i for i in 1..n])
```

$$\text{Type:} \quad \text{Void}$$

s 50

$$\frac{13943237577224054960759}{30990445042459967006400}$$

$$\text{Type:} \quad \text{Fraction Integer}$$

This function computes a Mersenne number, several of which are prime.

```
mersenne i == 2**i - 1
```

$$\text{Type:} \quad \text{Void}$$

If you type `mersenne`, Axiom shows you the function definition.

```
mersenne
```

$$mersenne\ i\ \ ==\ \ 2^i - 1$$

$$\text{Type:} \quad \text{FunctionCalled mersenne}$$

Generate a stream of Mersenne numbers.

```
[mersenne i for i in 1..]
```

$$[1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \ldots]$$

$$\text{Type:} \quad \text{Stream Integer}$$

Create a stream of those values of i such that `mersenne(i)` is prime.

```
mersenneIndex := [n for n in 1..  | prime?(mersenne(n))]
```

Compiling function mersenne with type PositiveInteger -> Integer

$$[2, 3, 5, 7, 13, 17, 19, 31, 61, 89, \ldots]$$

$$\text{Type:} \quad \text{Stream PositiveInteger}$$

Finally, write a function that returns the n-th Mersenne prime.

```
mersennePrime n == mersenne mersenneIndex(n)
```

<div align="right">Type: Void</div>

```
mersennePrime 5
```

<div align="center">8191</div>

<div align="right">Type: PositiveInteger</div>

6.6 Declared vs. Undeclared Functions

If you declare the type of a function, you can apply it to any data that can be converted to the source type of the function.

Define **f** with type Integer $\rightarrow$ Integer.

```
f(x:  Integer):  Integer == x + 1
```

> Function declaration f : Integer -> Integer has been added to
>   workspace.

<div align="right">Type: Void</div>

The function **f** can be applied to integers, . . .

```
f 9
```

> Compiling function f with type Integer -> Integer

<div align="center">10</div>

<div align="right">Type: PositiveInteger</div>

and to values that convert to integers, . . .

```
f(-2.0)
```

<div align="center">-1</div>

```
                                        Type:   Integer
```

but not to values that cannot be converted to integers.

```
f(2/3)
```

```
   Conversion failed in the compiled user function f .

   Cannot convert from type Fraction Integer to Integer for value
   2
   -
   3
```

To make the function over a wide range of types, do not declare its type. Give the same definition with no declaration.

```
g x == x + 1
```

```
                                           Type:   Void
```

If $x + 1$ makes sense, you can apply **g** to x.

```
g 9
```

```
   Compiling function g with type PositiveInteger -> PositiveInteger
```

```
                        10
```

```
                        Type:   PositiveInteger
```

A version of **g** with different argument types get compiled for each new kind of argument used.

```
g(2/3)
```

```
   Compiling function g with type Fraction Integer -> Fraction Integer
```

$$\frac{5}{3}$$

```
                        Type:   Fraction Integer
```

Here $x + 1$ for $x = "axiom"$ makes no sense.

```
g("axiom")
```

```
   There are 11 exposed and 5 unexposed library operations named +
      having 2 argument(s) but none was determined to be applicable.
      Use HyperDoc Browse, or issue
                              )display op +
      to learn more about the available operations. Perhaps
      package-calling the operation or using coercions on the arguments
      will allow you to apply the operation.
   Cannot find a definition or applicable library operation named +
      with argument type(s)
                                 String
                              PositiveInteger

      Perhaps you should use "@" to indicate the required return type,
      or "$" to specify which version of the function you need.
   AXIOM will attempt to step through and interpret the code.
   There are 11 exposed and 5 unexposed library operations named +
      having 2 argument(s) but none was determined to be applicable.
      Use HyperDoc Browse, or issue
                              )display op +
      to learn more about the available operations. Perhaps
      package-calling the operation or using coercions on the arguments
      will allow you to apply the operation.

   Cannot find a definition or applicable library operation named +
      with argument type(s)
                                 String
                              PositiveInteger

      Perhaps you should use "@" to indicate the required return type,
      or "$" to specify which version of the function you need.
```

As you will see in Chapter 12 on page 911, Axiom has a formal idea of categories for what "makes sense."

6.7   Functions vs. Operations

A function is an object that you can create, manipulate, pass to, and return from functions (for some interesting examples of library functions that manipulate functions, see 9.51 on page 692). Yet, we often seem to use the term *operation* and *function* interchangeably in Axiom. What is the distinction?

First consider values and types associated with some variable n in your workspace. You can make the declaration `n :  Integer`, then assign n an integer value. You then speak of the integer n. However, note that the integer is not the name n itself, but the value that you assign to n.

Similarly, you can declare a variable f in your workspace to have type Integer $\rightarrow$ Integer, then assign f, through a definition or an assignment of an anonymous function. You then speak of the function f. However, the function is not f, but the value that you assign to f.

A function is a value, in fact, some machine code for doing something. Doing what? Well, performing some *operation*. Formally, an operation consists of the constituent parts of f in your workspace, excluding the value; thus an operation has a name and a type. An operation is what domains and packages export. Thus `Ring` exports one operation "`+`". Every ring also exports this operation. Also, the author of every ring in the system is obliged under contract (see 11.3 on page 899) to provide an implementation for this operation.

This chapter is all about functions—how you create them interactively and how you apply them to meet your needs. In Chapter 11 on page 897 you will learn how to create them for the Axiom library. Then in Chapter 12 on page 911, you will learn about categories and exported operations.

6.8 Delayed Assignments vs. Functions with No Arguments

In 5.1 on page 195 we discussed the difference between immediate and delayed assignments. In this section we show the difference between delayed assignments and functions of no arguments.

A function of no arguments is sometimes called a *nullary function.*

```
sin24() == sin(24.0)
```

                                                      Type: Void

You must use the parentheses "()" to evaluate it. Like a delayed assignment, the right-hand-side of a function evaluation is not evaluated until the left-hand-side is used.

```
sin24()
```

```
   Compiling function sin24 with type () -> Float
```

$$-0.9055783620\ 0662384514$$

<div align="right">

`Type:  Float`

</div>

If you omit the parentheses, you just get the function definition.

`sin24`

$$sin24\,() \;\; == \;\; \sin{(24.0)}$$

<div align="right">

`Type:  FunctionCalled sin24`

</div>

You do not use the parentheses "()" in a delayed assignment...

`cos24 == cos(24.0)`

<div align="right">

`Type:  Void`

</div>

nor in the evaluation.

`cos24`

```
Compiling body of rule cos24 to compute value of type Float
```

<div align="center">

0.4241790073 3699697594

</div>

<div align="right">

`Type:  Float`

</div>

The only syntactic difference between delayed assignments and nullary functions is that you use "()" in the latter case.

6.9   How Axiom Determines What Function to Use

What happens if you define a function that has the same name as a library function? Well, if your function has the same name and number of arguments (we sometimes say *arity*) as another function in the library, then your function covers up the library function. If you want then to call the library function, you will have to *package-call* it. Axiom can use both the functions you write and those that come from the library. Let's do a simple example to illustrate this.

Suppose you (wrongly!) define **sin** in this way.

```
sin x == 1.0
```

<div align="right">Type: Void</div>

The value 1.0 is returned for any argument.

```
sin 4.3
```

```
   Compiling function sin with type Float -> Float
```

<div align="center">1.0</div>

<div align="right">Type: Float</div>

If you want the library operation, we have to package-call it (see 2.9 on page 162 for more information).

```
sin(4.3)$Float
```

$$-0.91616593674945498404$$

<div align="right">Type: Float</div>

```
sin(34.6)$Float
```

$$-0.042468034716950101543$$

<div align="right">Type: Float</div>

Even worse, say we accidentally used the same name as a library function in the function.

```
sin x == sin x
```

```
   Compiled code for sin has been cleared.
   1 old definition(s) deleted for function or rule sin
```

<div align="right">Type: Void</div>

Then Axiom definitely does not understand us.

```
sin 4.3
```

```
AXIOM cannot determine the type of sin because it cannot analyze
   the non-recursive part, if that exists. This may be remedied
   by declaring the function.
```

Again, we could package-call the inside function.

```
sin x == sin(x)$Float
```

```
   1 old definition(s) deleted for function or rule sin
```

<div align="right">Type: Void</div>

```
sin 4.3
```

```
   Compiling function sin with type Float -> Float
```

```
+++ |*1;sin;1;G82322| redefined
```

$$-0.91616593674945498404$$

<div align="right">Type: Float</div>

Of course, you are unlikely to make such obvious errors. It is more probable that you would write a function and in the body use a function that you think is a library function. If you had also written a function by that same name, the library function would be invisible.

How does Axiom determine what library function to call? It very much depends on the particular example, but the simple case of creating the polynomial $x + 2/3$ will give you an idea.

1. The x is analyzed and its default type is `Variable(x)`.

2. The 2 is analyzed and its default type is `PositiveInteger`.

3. The 3 is analyzed and its default type is `PositiveInteger`.

4. Because the arguments to "/" are integers, Axiom gives the expression $2/3$ a default target type of `Fraction(Integer)`.

5. Axiom looks in `PositiveInteger` for "/". It is not found.

6. Axiom looks in `Fraction(Integer)` for "/". It is found for arguments of type `Integer`.

7. The 2 and 3 are converted to objects of type `Integer` (this is trivial) and "/" is applied, creating an object of type `Fraction(Integer)`.

8. No "+" for arguments of types `Variable(x)` and `Fraction(Integer)` are found in either domain.

9. Axiom resolves (see 2.10 on page 166) the types and gets `Polynomial (Fraction (Integer))`.

10. The x and the 2/3 are converted to objects of this type and `+` is applied, yielding the answer, an object of type `Polynomial (Fraction (Integer))`.

6.10 Compiling vs. Interpreting

When possible, Axiom completely determines the type of every object in a function, then translates the function definition to Common Lisp or to machine code (see the next section). This translation, called compilation, happens the first time you call the function and results in a computational delay. Subsequent function calls with the same argument types use the compiled version of the code without delay.

If Axiom cannot determine the type of everything, the function may still be executed but in interpret-code mode: each statement in the function is analyzed and executed as the control flow indicates. This process is slower than executing a compiled function, but it allows the execution of code that may involve objects whose types change.

If Axiom decides that it cannot compile the code, it issues a message stating the problem and then the following message:

We will attempt to step through and interpret the code.

This is not a time to panic. Rather, it just means that what you gave to Axiom is somehow ambiguous: either it is not specific enough to be analyzed completely, or it is beyond Axiom's present interactive compilation abilities.

This function runs in interpret-code mode, but it does not compile.

```
varPolys(vars) ==
  for var in vars repeat
    output(1 :: UnivariatePolynomial(var,Integer))
```

                                                              Type: Void

For *vars* equal to $['x,' y,' z]$, this function displays 1 three times.

```
varPolys ['x,'y,'z]
```

```
Cannot compile conversion for types involving local variables.
   In particular, could not compile the expression involving ::
   UnivariatePolynomial(var,Integer)
 AXIOM will attempt to step through and interpret the code.
 1
 1
 1
```

                                                              Type: Void

The type of the argument to **output** changes in each iteration, so Axiom cannot compile the function. In this case, even the inner loop by itself would have a problem:

```
for var in ['x,'y,'z] repeat
  output(1 :: UnivariatePolynomial(var,Integer))
```

```
Cannot compile conversion for types involving local variables.
   In particular, could not compile the expression involving ::
   UnivariatePolynomial(var,Integer)
 AXIOM will attempt to step through and interpret the code.
 1
 1
 1
```

                                                              Type: Void

Sometimes you can help a function to compile by using an extra conversion or by using *pretend*. See 2.8 on page 158 for details.

When a function is compilable, you have the choice of whether it is compiled to Common Lisp and then interpreted by the Common Lisp interpreter or then further compiled from Common Lisp to machine code. The option is controlled via `)set functions compile`. Issue `)set functions compile on` to compile all the way to machine code. With the default setting `)set functions compile off`, Axiom has its Common Lisp code interpreted because the overhead of further compilation is larger than the run-time of most of the functions our users have defined. You may find that selectively turning this option on and off will give you the best performance in your particular application. For example, if you are writing functions for graphics applications where hundreds of points are being computed, it is almost certainly true that you will get the best performance by issuing `)set functions compile on`.

6.11   Piece-Wise Function Definitions

To move beyond functions defined in one line, we introduce in this section functions that are defined piece-by-piece. That is, we say "use this definition when the argument is such-and-such and use this other definition when the argument is that-and-that."

6.11.1   A Basic Example

There are many other ways to define a factorial function for nonnegative integers. You might say factorial of 0 is 1, otherwise factorial of n is n times factorial of $n - 1$. Here is one way to do this in Axiom.

Here is the value for $n = 0$.

```
fact(0) == 1
```

                                                            Type: Void

Here is the value for $n > 0$. The vertical bar "|" means "such that".

```
fact(n | n > 0) == n * fact(n - 1)
```

                                                            Type: Void

What is the value for $n = 3$?

```
fact(3)
```

```
   Compiling function fact with type Integer -> Integer
   Compiling function fact as a recurrence relation.
```

                                        6

                                            Type: PositiveInteger

What is the value for $n = -3$?

```
fact(-3)
```

```
   You did not define fact for argument -3 .
```

Now for a second definition. Here is the value for $n = 0$.

```
facto(0) == 1
```

<div align="right">Type: Void</div>

Give an error message if $n < 0$.

```
facto(n | n < 0) == error "arguments to facto must be
non-negative"
```

<div align="right">Type: Void</div>

Here is the value otherwise.

```
facto(n) == n * facto(n - 1)
```

<div align="right">Type: Void</div>

What is the value for $n = 7$?

```
facto(3)
```

```
   Compiling function facto with type Integer -> Integer
```

<div align="center">6</div>

<div align="right">Type: PositiveInteger</div>

What is the value for $n = -7$?

```
facto(-7)
```

```
   Error signalled from user code in function facto:
      arguments to facto must be non-negative
```

<div align="right">Type: PositiveInteger</div>

To see the current piece-wise definition of a function, use)display value.

```
)display value facto
```

```
   Definition:
     facto 0 == 1
     facto (n | n < 0) ==
       error(arguments to facto must be non-negative)
     facto n == n facto(n - 1)
```

In general a *piece-wise definition* of a function consists of two or more parts. Each part gives a "piece" of the entire definition. Axiom collects the pieces of a function as you enter them. When you ask for a value of the function, it then "glues" the pieces together to form a function.

The two piece-wise definitions for the factorial function are examples of recursive functions, that is, functions that are defined in terms of themselves. Here is an interesting doubly-recursive function. This function returns the value 11 for all positive integer arguments.

Here is the first of two pieces.

```
eleven(n | n < 1) == n + 11
```

<div align="right">Type: Void</div>

And the general case.

```
eleven(m) == eleven(eleven(m - 12))
```

<div align="right">Type: Void</div>

Compute *elevens*, the infinite stream of values of *eleven*.

```
elevens := [eleven(i) for i in 0..]
```

$$[11, 11, 11, 11, 11, 11, 11, 11, 11, 11, \ldots]$$

<div align="right">Type: Stream Integer</div>

What is the value at $n = 200$?

```
elevens 200
```

<div align="center">11</div>

<div align="right">

Type: PositiveInteger

</div>

What is the Axiom's definition of *eleven*?

```
)display value eleven
```

```
  Definition:
    eleven (m | m < 1) == m + 11
    eleven m == eleven(eleven(m - 12))
```

6.11.2   Picking Up the Pieces

Here are the details about how Axiom creates a function from its pieces. Axiom converts the *i*-th piece of a function definition into a conditional expression of the form: `if` $pred_i$ `then` $expression_i$. If any new piece has a $pred_i$ that is identical (after all variables are uniformly named) to an earlier $pred_j$, the earlier piece is removed. Otherwise, the new piece is always added at the end.

If there are *n* pieces to a function definition for *f*, the function defined *f* is:

       `if` $pred_1$ `then` $expression_1$ `else`

          . . .

       `if` $pred_n$ `then` $expression_n$ `else`

       `error "You did not define f for argument <arg>."`

You can give definitions of any number of mutually recursive function definitions, piece-wise or otherwise. No computation is done until you ask for a value. When you do ask for a value, all the relevant definitions are gathered, analyzed, and translated into separate functions and compiled.

Let's recall the definition of **eleven** from the previous section.

```
eleven(n | n < 1) == n + 11
```

<div align="right">

Type: Void

</div>

```
eleven(m) == eleven(eleven(m - 12))
```

<div align="right">

Type: Void

</div>

A similar doubly-recursive function below produces −11 for all negative positive integers. If you haven't worked out why or how **eleven** works, the structure of this definition gives a clue.

This definition we write as a block.

```
minusEleven(n) ==
  n >= 0 => n - 11
  minusEleven (5 + minusEleven(n + 7))
```

<div align="right">Type: Void</div>

Define $s(n)$ to be the sum of plus and minus "eleven" functions divided by n. Since $11 - 11 = 0$, we define $s(0)$ to be 1.

```
s(0) == 1
```

<div align="right">Type: Void</div>

And the general term.

```
s(n) == (eleven(n) + minusEleven(n))/n
```

<div align="right">Type: Void</div>

What are the first ten values of s?

```
[s(n) for n in 0..]
```

$$[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, \ldots]$$

<div align="right">Type: Stream Fraction Integer</div>

Axiom can create infinite streams in the positive direction (for example, for index values $0, 1, \ldots$) or negative direction (for example, for $0, -1, -2, \ldots$). Here we would like a stream of values of $s(n)$ that is infinite in both directions. The function $t(n)$ below returns the n-th term of the infinite stream

$$[s(0), s(1), s(-1), s(2), s(-2), \ldots]$$

Its definition has three pieces.

Define the initial term.

```
t(1) == s(0)
```

<div align="right">Type: Void</div>

The even numbered terms are the $s(i)$ for positive i. We use "quo" rather than "/" since we want the result to be an integer.

```
t(n | even?(n)) == s(n quo 2)
```

<div align="right">Type: Void</div>

Finally, the odd numbered terms are the $s(i)$ for negative i. In piece-wise definitions, you can use different variables to define different pieces. Axiom will not get confused.

```
t(p) == s(- p quo 2)
```

<div align="right">Type: Void</div>

Look at the definition of t. In the first piece, the variable n was used; in the second piece, p. Axiom always uses your last variable to display your definitions back to you.

```
)display value t
```

```
   Definition:
     t 1 == s(0)
     t (p | even?(p)) == s(p quo 2)
     t p == s(- p quo 2)
```

Create a series of values of s applied to alternating positive and negative arguments.

```
[t(i) for i in 1..]
```

```
   Compiling function s with type Integer -> Fraction Integer
   Compiling function t with type PositiveInteger -> Fraction Integer
```

$$[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, \ldots]$$

<div align="right">Type: Stream Fraction Integer</div>

Evidently $t(n) = 1$ for all i. Check it at $n = 100$.

```
t(100)
```

<div align="center">1</div>

<div align="right">Type: Fraction Integer</div>

6.11.3 Predicates

We have already seen some examples of predicates (6.11.1 on page 249). Predicates are `Boolean`-valued expressions and Axiom uses them for filtering collections (see 5.5 on page 224) and for placing constraints on function arguments. In this section we discuss their latter usage.

The simplest use of a predicate is one you don't see at all.

```
opposite 'right == 'left
```

```
                                                    Type:  Void
```

Here is a longer way to give the "opposite definition."

```
opposite (x | x = 'left) == 'right
```

```
                                                    Type:  Void
```

Try it out.

```
for x in ['right,'left,'inbetween] repeat output opposite x
```

```
Compiling function opposite with type
   OrderedVariableList [right, left,inbetween] -> Symbol
 left
 right
```

```
The function opposite is not defined for the given argument(s).
```

Explicit predicates tell Axiom that the given function definition piece is to be applied if the predicate evaluates to `true` for the arguments to the function. You can use such "constant" arguments for integers, strings, and quoted symbols. The `Boolean` values `true` and `false` can also be used if qualified with "@" or "$" and `Boolean`. The following are all valid function definition fragments using constant arguments.

```
a(1) == ...
b("unramified") == ...
c('untested) == ...
d(true@Boolean) == ...
```

If a function has more than one argument, each argument can have its own predicate. However, if a predicate involves two or more arguments, it must be given *after* all the arguments mentioned in the predicate have been given. You are always safe to give a single predicate at the end of the argument list.

A function involving predicates on two arguments.

```
inFirstHalfQuadrant(x | x > 0,y | y < x) == true
```

<div align="right">Type: Void</div>

This is incorrect as it gives a predicate on y before the argument y is given.

```
inFirstHalfQuadrant(x | x > 0 and y < x,y) == true
```

   1 old definition(s) deleted for function or rule inFirstHalfQuadrant

<div align="right">Type: Void</div>

It is always correct to write the predicate at the end.

```
inFirstHalfQuadrant(x,y | x > 0 and y < x) == true
```

   1 old definition(s) deleted for function or rule inFirstHalfQuadrant

<div align="right">Type: Void</div>

Here is the rest of the definition.

```
inFirstHalfQuadrant(x,y) == false
```

<div align="right">Type: Void</div>

Try it out.

```
[inFirstHalfQuadrant(i,3) for i in 1..5]
```

   Compiling function inFirstHalfQuadrant with type (PositiveInteger,
      PositiveInteger) -> Boolean

$$[\text{false}, \text{false}, \text{false}, \text{true}, \text{true}]$$

<div align="right">Type: List Boolean</div>

6.12   Caching Previously Computed Results

By default, Axiom does not save the values of any function. You can cause it
to save values and not to recompute unnecessarily by using `)set functions`
`cache`. This should be used before the functions are defined or, at least, before
they are executed. The word following "cache" should be 0 to turn off caching,
a positive integer n to save the last n computed values or "all" to save all
computed values. If you then give a list of names of functions, the caching
only affects those functions. Use no list of names or "all" when you want to
define the default behavior for functions not specifically mentioned in other
`)set functions cache` statements. If you give no list of names, all functions
will have the caching behavior. If you explicitly turn on caching for one or more
names, you must explicitly turn off caching for those names when you want to
stop saving their values.

This causes the functions **f** and **g** to have the last three computed values saved.

```
)set functions cache 3 f g
```

```
   function f will cache the last 3 values.
   function g will cache the last 3 values.
```

This is a sample definition for **f**.

```
f x == factorial(2**x)
```

<div align="right">Type: Void</div>

A message is displayed stating what **f** will cache.

```
f(4)
```

```
   Compiling function f with type PositiveInteger -> Integer
   f will cache 3 most recently computed value(s).
```

```
+++ |*1;f;1;G82322| redefined
```

<div align="center">20922789888000</div>

<div align="right">Type: PositiveInteger</div>

This causes all other functions to have all computed values saved by default.

```
)set functions cache all
```

```
    In general, interpreter functions will cache all values.
```

This causes all functions that have not been specifically cached in some way to have no computed values saved.

```
)set functions cache 0
```

```
 In general, functions will cache no returned values.
```

We also make **f** and **g** uncached.

```
)set functions cache 0 f g
```

```
   Caching for function f is turned off
   Caching for function g is turned off
```

Be careful about caching functions that have side effects. Such a function might destructively modify the elements of an array or issue a **draw** command, for example. A function that you expect to execute every time it is called should not be cached. Also, it is highly unlikely that a function with no arguments should be cached.

You should also be careful about caching functions that depend on free variables. See 6.16 on page 269 for an example.

6.13   Recurrence Relations

One of the most useful classes of function are those defined via a "recurrence relation." A *recurrence relation* makes each successive value depend on some or all of the previous values. A simple example is the ordinary "factorial" function:

```
fact(0) == 1
fact(n | n > 0) == n * fact(n-1)
```

The value of $fact(10)$ depends on the value of $fact(9)$, $fact(9)$ on $fact(8)$, and so on. Because it depends on only one previous value, it is usually called a *first order recurrence relation.* You can easily imagine a function based on two, three or more previous values. The Fibonacci numbers are probably the most famous function defined by a second order recurrence relation.

The library function **fibonacci** computes Fibonacci numbers. It is obviously optimized for speed.

```
[fibonacci(i) for i in 0..]
```

$$[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots]$$

<div align="right">Type: Stream Integer</div>

Define the Fibonacci numbers ourselves using a piece-wise definition.

```
fib(1) == 1
```

<div align="right">Type: Void</div>

```
fib(2) == 1
```

<div align="right">Type: Void</div>

```
fib(n) == fib(n-1) + fib(n-2)
```

<div align="right">Type: Void</div>

As defined, this recurrence relation is obviously doubly-recursive. To compute $fib(10)$, we need to compute $fib(9)$ and $fib(8)$. And to $fib(9)$, we need to compute $fib(8)$ and $fib(7)$. And so on. It seems that to compute $fib(10)$ we need to compute $fib(9)$ once, $fib(8)$ twice, $fib(7)$ three times. Look familiar? The number of function calls needed to compute *any* second order recurrence relation in the obvious way is exactly $fib(n)$. These numbers grow! For example, if Axiom actually did this, then $fib(500)$ requires more than 10^{104} function calls. And, given all this, our definition of **fib** obviously could not be used to calculate the five-hundredth Fibonacci number.

Let's try it anyway.

```
fib(500)
```

```
   Compiling function fib with type Integer -> PositiveInteger
   Compiling function fib as a recurrence relation.
```

139423224561697880139724382870407283950070256587697307264 10_
8962948325571622863290691557658876222521294125

<div align="right">Type: PositiveInteger</div>

Since this takes a short time to compute, it obviously didn't do as many as 10^{104} operations! By default, Axiom transforms any recurrence relation it recognizes into an iteration. Iterations are efficient. To compute the value of the n-th term of a recurrence relation using an iteration requires only n function calls. Note that if you compare the speed of our **fib** function to the library function, our version is still slower. This is because the library **fibonacci** uses a "powering algorithm" with a computing time proportional to $\log^3(n)$ to compute `fibonacci(n)`.

To turn off this special recurrence relation compilation, issue

```
)set functions recurrence off
```

To turn it back on, substitute "`on`" for "`off`".

The transformations that Axiom uses for **fib** caches the last two values. For a more general k-th order recurrence relation, Axiom caches the last k values. If, after computing a value for **fib**, you ask for some larger value, Axiom picks up the cached values and continues computing from there. See 6.16 on page 269 for an example of a function definition that has this same behavior. Also see 6.12 on page 257 for a more general discussion of how you can cache function values.

Recurrence relations can be used for defining recurrence relations involving polynomials, rational functions, or anything you like. Here we compute the infinite stream of Legendre polynomials.

The Legendre polynomial of degree 0.

```
p(0) == 1
```

                                                                    Type: Void

The Legendre polynomial of degree 1.

```
p(1) == x
```

                                                                    Type: Void

The Legendre polynomial of degree n.

```
p(n) == ((2*n-1)*x*p(n-1) - (n-1)*p(n-2))/n
```

                                                                    Type: Void

Compute the Legendre polynomial of degree 6.

```
p(6)
```

```
    Compiling function p with type Integer -> Polynomial Fraction
        Integer
    Compiling function p as a recurrence relation.
```

$$\frac{231}{16}\,x^6 - \frac{315}{16}\,x^4 + \frac{105}{16}\,x^2 - \frac{5}{16}$$

<div align="right">Type: Polynomial Fraction Integer</div>

6.14 Making Functions from Objects

There are many times when you compute a complicated expression and then wish to use that expression as the body of a function. Axiom provides an operation called **function** to do this. It creates a function object and places it into the workspace. There are several versions, depending on how many arguments the function has. The first argument to **function** is always the expression to be converted into the function body, and the second is always the name to be used for the function. For more information, see section 9.50 on page 689.

Start with a simple example of a polynomial in three variables.

```
p := -x + y**2 - z**3
```

$$-z^3 + y^2 - x$$

<div align="right">Type: Polynomial Integer</div>

To make this into a function of no arguments that simply returns the polynomial, use the two argument form of **function**.

```
function(p,'f0)
```

$$f0$$

<div align="right">Type: Symbol</div>

To avoid possible conflicts (see below), it is a good idea to quote always this second argument.

```
f0
```

$$f0 \; () \;\; == \;\; -z^3 + y^2 - x$$

<div align="right">Type: FunctionCalled f0</div>

This is what you get when you evaluate the function.

f0()

$$-z^3 + y^2 - x$$

<div align="right">Type: Polynomial Integer</div>

To make a function in x, use a version of **function** that takes three arguments. The last argument is the name of the variable to use as the parameter. Typically, this variable occurs in the expression and, like the function name, you should quote it to avoid possible confusion.

function(p,'f1,'x)

$$f1$$

<div align="right">Type: Symbol</div>

This is what the new function looks like.

f1

$$f1 \; x \;\; == \;\; -z^3 + y^2 - x$$

<div align="right">Type: FunctionCalled f1</div>

This is the value of **f1** at $x = 3$. Notice that the return type of the function is Polynomial (Integer), the same as p.

f1(3)

> Compiling function f1 with type PositiveInteger -> Polynomial
>   Integer

$$-z^3 + y^2 - 3$$

<div align="right">Type: Polynomial Integer</div>

To use x and y as parameters, use the four argument form of **function**.

```
function(p,'f2,'x,'y)
```

$$f2$$

<div align="right">Type: Symbol</div>

f2

$$f2\ (x,y)\ ==\ -z^3 + y^2 - x$$

<div align="right">Type: FunctionCalled f2</div>

Evaluate $f2$ at $x = 3$ and $y = 0$. The return type of **f2** is still `Polynomial(Integer)` because the variable z is still present and not one of the parameters.

```
f2(3,0)
```

$$-z^3 - 3$$

<div align="right">Type: Polynomial Integer</div>

Finally, use all three variables as parameters. There is no five argument form of **function**, so use the one with three arguments, the third argument being a list of the parameters.

```
function(p,'f3,['x,'y,'z])
```

$$f3$$

<div align="right">Type: Symbol</div>

Evaluate this using the same values for x and y as above, but let z be -6. The result type of **f3** is `Integer`.

f3

$$f3\ (x,y,z)\ ==\ -z^3 + y^2 - x$$

<div align="right">Type: FunctionCalled f3</div>

```
f3(3,0,-6)
```

```
Compiling function f3 with type (PositiveInteger,NonNegativeInteger,
    Integer) -> Integer
```

$$213$$

                                          Type: PositiveInteger

The four functions we have defined via p have been undeclared. To declare a function whose body is to be generated by **function**, issue the declaration *before* the function is created.

```
g:   (Integer, Integer) -> Float
```

                                                  Type: Void

```
D(sin(x-y)/cos(x+y),x)
```

$$\frac{-\sin\left(y-x\right)\,\sin\left(y+x\right)+\cos\left(y-x\right)\,\cos\left(y+x\right)}{\cos\left(y+x\right)^2}$$

                                          Type: Expression Integer

```
function(%,'g,'x,'y)
```

$$g$$

                                                Type: Symbol

```
g
```

$$g\left(x,y\right)\ ==\ \frac{-\sin\left(y-x\right)\,\sin\left(y+x\right)+\cos\left(y-x\right)\,\cos\left(y+x\right)}{\cos\left(y+x\right)^2}$$

                                          Type: FunctionCalled g

It is an error to use g without the quote in the penultimate expression since g had been declared but did not have a value. Similarly, since it is common to overuse variable names like x, y, and so on, you avoid problems if you always quote the variable names for **function**. In general, if x has a value and you use x without a quote in a call to **function**, then Axiom does not know what you are trying to do.

What kind of object is allowable as the first argument to **function**? Let's use
the Browse facility of HyperDoc to find out. At the main Browse menu, enter the
string `function` and then click on **Operations.** The exposed operations called
function all take an object whose type belongs to category `ConvertibleTo`
`InputForm`. What domains are those? Go back to the main Browse menu, erase
`function`, enter `ConvertibleTo` in the input area, and click on **categories** on
the `Constructors` line. At the bottom of the page, enter `InputForm` in the input
area following **S =**. Click on `Cross Reference` and then on `Domains`. The list
you see contains over forty domains that belong to the category `ConvertibleTo`
`InputForm`. Thus you can use **function** for `Integer`, `Float`, `String`, `Complex`,
`Expression`, and so on.

6.15 Functions Defined with Blocks

You need not restrict yourself to functions that only fit on one line or are written
in a piece-wise manner. The body of the function can be a block, as discussed
in 5.2 on page 199.

Here is a short function that swaps two elements of a list, array or vector.

```
swap(m,i,j) ==
  temp := m.i
  m.i := m.j
  m.j := temp
```

<div align="right">Type: Void</div>

The significance of **swap** is that it has a destructive effect on its first argument.

```
k := [1,2,3,4,5]
```

$$[1, 2, 3, 4, 5]$$

<div align="right">Type: List PositiveInteger</div>

```
swap(k,2,4)
```

```
   Compiling function swap with type (List PositiveInteger,
      PositiveInteger,PositiveInteger) -> PositiveInteger
```

<div align="center">2</div>

<div align="right">Type: PositiveInteger</div>

You see that the second and fourth elements are interchanged.

```
k
```

$$[1, 4, 3, 2, 5]$$

<div align="right">Type: List PositiveInteger</div>

Using this, we write a couple of different sort functions. First, a simple bubble sort. The operation "#" returns the number of elements in an aggregate.

```
bubbleSort(m) ==
  n := #m
  for i in 1..(n-1) repeat
    for j in n..(i+1) by -1 repeat
      if m.j < m.(j-1) then swap(m,j,j-1)
  m
```

<div align="right">Type: Void</div>

Let this be the list we want to sort.

```
m := [8,4,-3,9]
```

$$[8, 4, -3, 9]$$

<div align="right">Type: List Integer</div>

This is the result of sorting.

```
bubbleSort(m)
```

```
   Compiling function swap with type (List Integer,Integer,Integer) ->
      Integer
```

```
+++ |*3;swap;1;G82322| redefined
   Compiling function bubbleSort with type List Integer -> List Integer
```

$$[-3, 4, 8, 9]$$

<div align="right">Type: List Integer</div>

Moreover, m is destructively changed to be the sorted version.

```
m
```

$$[-3, 4, 8, 9]$$

<div align="right">Type: List Integer</div>

This function implements an insertion sort. The basic idea is to traverse the list and insert the i-th element in its correct position among the $i-1$ previous elements. Since we start at the beginning of the list, the list elements before the i-th element have already been placed in ascending order.

```
insertionSort(m) ==
  for i in 2..#m repeat
    j := i
    while j > 1 and m.j < m.(j-1) repeat
      swap(m,j,j-1)
      j := j - 1
  m
```

<div align="right">Type: Void</div>

As with our bubble sort, this is a destructive function.

```
m := [8,4,-3,9]
```

$$[8, 4, -3, 9]$$

<div align="right">Type: List Integer</div>

```
insertionSort(m)
```

```
   Compiling function insertionSort with type List Integer -> List
      Integer
```

$$[-3, 4, 8, 9]$$

<div align="right">Type: List Integer</div>

```
m
```

$$[-3, 4, 8, 9]$$

<div align="right">Type: List Integer</div>

Neither of the above functions is efficient for sorting large lists since they reference elements by asking for the j-th element of the structure m.

Here is a more efficient bubble sort for lists.

```
bubbleSort2(m: List Integer): List Integer ==
  null m => m
  l := m
  while not null (r := l.rest) repeat
    r := bubbleSort2 r
    x := l.first
    if x < r.first then
      l.first := r.first
      r.first := x
    l.rest := r
    l := l.rest
  m
```

```
  Function declaration bubbleSort2 : List Integer -> List Integer has
    been added to workspace.
```

<div align="right">Type: Void</div>

Try it out.

```
bubbleSort2 [3,7,2]
```

$$[7, 3, 2]$$

<div align="right">Type: List Integer</div>

This definition is both recursive and iterative, and is tricky! Unless you are *really* curious about this definition, we suggest you skip immediately to the next section.

Here are the key points in the definition. First notice that if you are sorting a list with less than two elements, there is nothing to do: just return the list. This definition returns immediately if there are zero elements, and skips the entire `while` loop if there is just one element.

The second point to realize is that on each outer iteration, the bubble sort ensures that the minimum element is propagated leftmost. Each iteration of the `while` loop calls **bubbleSort2** recursively to sort all but the first element. When finished, the minimum element is either in the first or second position. The conditional expression ensures that it comes first. If it is in the second, then a swap occurs. In any case, the **rest** of the original list must be updated to hold the result of the recursive call.

6.16 Free and Local Variables

When you want to refer to a variable that is not local to your function, use a "`free`" declaration. Variables declared to be `free` are assumed to be defined globally in the workspace.

This is a global workspace variable.

```
counter := 0
```

$$0$$

                                        Type: NonNegativeInteger

This function refers to the global *counter*.

```
f() ==
  free counter
  counter := counter + 1
```

                                                Type: Void

The global *counter* is incremented by 1.

```
f()
```

   Compiling function f with type () -> NonNegativeInteger

+++ |*0;f;1;G82322| redefined

$$1$$

                                        Type: PositiveInteger

```
counter
```

$$1$$

                                        Type: NonNegativeInteger

Usually Axiom can tell that you mean to refer to a global variable and so `free` isn't always necessary. However, for clarity and the sake of self-documentation, we encourage you to use it.

Declare a variable to be "`local`" when you do not want to refer to a global variable by the same name.

This function uses *counter* as a local variable.

```
g() ==
  local counter
  counter := 7
```

<div align="right">Type: Void</div>

Apply the function.

```
g()
```

<div align="center">7</div>

<div align="right">Type: PositiveInteger</div>

Check that the global value of *counter* is unchanged.

```
counter
```

<div align="center">1</div>

<div align="right">Type: NonNegativeInteger</div>

Parameters to a function are local variables in the function. Even if you issue a `free` declaration for a parameter, it is still local.

What happens if you do not declare that a variable x in the body of your function is `local` or `free`? Well, Axiom decides on this basis:

1. Axiom scans your function line-by-line, from top-to-bottom. The right-hand side of an assignment is looked at before the left-hand side.

2. If x is referenced before it is assigned a value, it is a `free` (global) variable.

3. If x is assigned a value before it is referenced, it is a `local` variable.

Set two global variables to 1.

```
a := b := 1
```

<div align="center">1</div>

<div align="right">Type: PositiveInteger</div>

Refer to a before it is assigned a value, but assign a value to b before it is referenced.

```
h() ==
  b := a + 1
  a := b + a
```

<div align="right">Type: Void</div>

Can you predict this result?

```
h()
```

```
   Compiling function h with type () -> PositiveInteger
```

```
+++ |*0;h;1;G82322| redefined
```

<div align="center">3</div>

<div align="right">Type: PositiveInteger</div>

How about this one?

```
[a, b]
```

$$[3, 1]$$

<div align="right">Type: List PositiveInteger</div>

What happened? In the first line of the function body for h, a is referenced on the right-hand side of the assignment. Thus a is a free variable. The variable b is not referenced in that line, but it is assigned a value. Thus b is a local variable and is given the value $a + 1 = 2$. In the second line, the free variable a is assigned the value $b + a$ which equals $2 + 1 = 3$. This is the value returned by the function. Since a was free in **h**, the global variable a has value 3. Since b was local in **h**, the global variable b is unchanged—it still has the value 1.

It is good programming practice always to declare global variables. However, by far the most common situation is to have local variables in your functions. No declaration is needed for this situation, but be sure to initialize their values.

Be careful if you use free variables and you cache the value of your function (see 6.12 on page 257). Caching *only* checks if the values of the function arguments are the same as in a function call previously seen. It does not check if any of the free variables on which the function depends have changed between function calls.

Turn on caching for **p**.

```
)set fun cache all p
```

    function p will cache all values.

Define **p** to depend on the free variable N.

```
p(i,x) == ( free N; reduce( + , [ (x-i)**n for n in 1..N ] ) )
```

                                                Type: Void

Set the value of N.

```
N := 1
```

                                1

                                        Type: PositiveInteger

Evaluate **p** the first time.

```
p(0, x)
```

                                x

                                    Type: Polynomial Integer

Change the value of N.

```
N := 2
```

                                2

                                        Type: PositiveInteger

Evaluate **p** the second time.

```
p(0, x)
```

                                x

                                    Type: Polynomial Integer

If caching had been turned off, the second evaluation would have reflected the changed value of N.

Turn off caching for **p**.

```
)set fun cache 0 p
```

```
   Caching for function p is turned off
```

Axiom does not allow *fluid variables*, that is, variables bound by a function f that can be referenced by functions called by f.

Values are passed to functions by *reference*: a pointer to the value is passed rather than a copy of the value or a pointer to a copy.

This is a global variable that is bound to a record object.

```
r :  Record(i :  Integer) := [1]
```

$$[i = 1]$$

```
                         Type:  Record(i:  Integer)
```

This function first modifies the one component of its record argument and then rebinds the parameter to another record.

```
resetRecord rr ==
  rr.i := 2
  rr := [10]
```

```
                                         Type:  Void
```

Pass r as an argument to **resetRecord**.

```
resetRecord r
```

$$[i = 10]$$

```
                         Type:  Record(i:  Integer)
```

The value of r was changed by the expression $rr.i := 2$ but not by $rr := [10]$.

```
r
```

$$[i = 2]$$

```
                                            Type:  Record(i:  Integer)
```

To conclude this section, we give an iterative definition of a function that computes Fibonacci numbers. This definition approximates the definition into which Axiom transforms the recurrence relation definition of **fib** in 6.13 on page 258.

Global variables `past` and `present` are used to hold the last computed Fibonacci numbers.

```
past := present := 1
```

$$1$$

```
                                            Type:  PositiveInteger
```

Global variable *index* gives the current index of *present*.

```
index := 2
```

$$2$$

```
                                            Type:  PositiveInteger
```

Here is a recurrence relation defined in terms of these three global variables.

```
fib(n) ==
  free past, present, index
  n < 3 => 1
  n = index - 1 => past
  if n < index-1 then
    (past,present) := (1,1)
    index := 2
  while (index < n) repeat
    (past,present) := (present, past+present)
    index := index + 1
  present
```

```
                                                       Type:  Void
```

Compute the infinite stream of Fibonacci numbers.

```
fibs := [fib(n) for n in 1..]
```

$$[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots]$$

<div align="right">Type: Stream PositiveInteger</div>

What is the 1000th Fibonacci number?

```
fibs 1000
```

```
43466557686937456435688527675040625802564660517371780402_
  8172908953655541794905189040387984007925516929592259308_
  0322634775209689623239873322471161642996440906533187938_
  29896964992851600370447613779516684922887 5
```

<div align="right">Type: PositiveInteger</div>

As an exercise, we suggest you write a function in an iterative style that computes the value of the recurrence relation $p(n) = p(n-1) - 2\,p(n-2) + 4\,p(n-3)$ having the initial values $p(1) = 1$, $p(2) = 3$ and $p(3) = 9$. How would you write the function using an element `OneDimensionalArray` or `Vector` to hold the previously computed values?

6.17 Anonymous Functions

An *anonymous function* is a function that is defined by giving a list of parameters, the "maps-to" compound symbol "`+->`" (from the mathematical symbol $\mapsto$), and by an expression involving the parameters, the evaluation of which determines the return value of the function.

$$(\ parm_1,\ parm_2,\ \ldots,\ parm_N\ )\ \texttt{+->}\ expression$$

You can apply an anonymous function in several ways.

1. Place the anonymous function definition in parentheses directly followed by a list of arguments.

2. Assign the anonymous function to a variable and then use the variable name when you would normally use a function name.

3. Use "`==`" to use the anonymous function definition as the arguments and body of a regular function definition.

4. Have a named function contain a declared anonymous function and use the result returned by the named function.

6.17.1 Some Examples

Anonymous functions are particularly useful for defining functions "on the fly." That is, they are handy for simple functions that are used only in one place. In the following examples, we show how to write some simple anonymous functions.

This is a simple absolute value function.

```
x +-> if x < 0 then -x else x
```

$$x \mapsto if\ x < 0 \quad \begin{array}{l} then\ -x \\ else\ x \end{array}$$

<div align="right">Type: AnonymousFunction</div>

```
abs1 := %
```

$$x \mapsto if\ x < 0 \quad \begin{array}{l} then\ -x \\ else\ x \end{array}$$

<div align="right">Type: AnonymousFunction</div>

This function returns **true** if the absolute value of the first argument is greater than the absolute value of the second, **false** otherwise.

```
(x,y) +-> abs1(x) > abs1(y)
```

$$(x, y) \mapsto abs1\,(y) < abs1\,(x)$$

<div align="right">Type: AnonymousFunction</div>

We use the above function to "sort" a list of integers.

```
sort(%,[3,9,-4,10,-3,-1,-9,5])
```

$$[10, -9, 9, 5, -4, -3, 3, -1]$$

<div align="right">Type: List Integer</div>

This function returns 1 if $i + j$ is even, -1 otherwise.

```
ev := ( (i,j) +-> if even?(i+j) then 1 else -1)
```

$$(i, j) \mapsto if\ even?\,(i + j) \quad \begin{array}{l} then\ 1 \\ else\ -1 \end{array}$$

We create a four-by-four matrix containing 1 or -1 depending on whether the row plus the column index is even or not.

```
matrix([ [ev(row,col) for row in 1..4] for col in 1..4])
```

$$\begin{bmatrix} 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{bmatrix}$$

                                          Type: Matrix Integer

This function returns `true` if a polynomial in x has multiple roots, `false` otherwise. It is defined and applied in the same expression.

```
( p +-> not one?(gcd(p,D(p,x))) )(x**2+4*x+4)
```

                            true

                                          Type: Boolean

This and the next expression are equivalent.

```
g(x,y,z) == cos(x + sin(y + tan(z)))
```

                                          Type: Void

The one you use is a matter of taste.

```
g == (x,y,z) +-> cos(x + sin(y + tan(z)))
```

   1 old definition(s) deleted for function or rule g

                                          Type: Void

6.17.2   Declaring Anonymous Functions

If you declare any of the arguments you must declare all of them. Thus,

```
(x: INT,y): FRAC INT +-> (x + 2*y)/(y - 1)
```

is not legal.

This is an example of a fully declared anonymous function. The output shown just indicates that the object you created is a particular kind of map, that is, function.

```
(x:  INT,y:  INT): FRAC INT +-> (x + 2*y)/(y - 1)
```

<div align="center">theMap(...)</div>

<div align="center">Type: ((Integer,Integer) -> Fraction Integer)</div>

Axiom allows you to declare the arguments and not declare the return type.

```
(x:  INT,y:  INT) +-> (x + 2*y)/(y - 1)
```

<div align="center">theMap(...)</div>

<div align="center">Type: ((Integer,Integer) -> Fraction Integer)</div>

The return type is computed from the types of the arguments and the body of the function. You cannot declare the return type if you do not declare the arguments. Therefore,

```
(x,y): FRAC INT +-> (x + 2*y)/(y - 1)
```

is not legal. This and the next expression are equivalent.

```
h(x:  INT,y:  INT): FRAC INT == (x + 2*y)/(y - 1)
```

```
   Function declaration h : (Integer,Integer) -> Fraction Integer
      has been added to workspace.
```

<div align="right">Type: Void</div>

The one you use is a matter of taste.

```
h == (x:  INT,y:  INT): FRAC INT +-> (x + 2*y)/(y - 1)
```

```
   Function declaration h : (Integer,Integer) -> Fraction Integer
      has been added to workspace.
   1 old definition(s) deleted for function or rule h
```

<div align="right">Type: Void</div>

When should you declare an anonymous function?

1. If you use an anonymous function and Axiom can't figure out what you are trying to do, declare the function.

2. If the function has nontrivial argument types or a nontrivial return type that Axiom may be able to determine eventually, but you are not willing to wait that long, declare the function.

3. If the function will only be used for arguments of specific types and it is not too much trouble to declare the function, do so.

4. If you are using the anonymous function as an argument to another function (such as **map** or **sort**), consider declaring the function.

5. If you define an anonymous function inside a named function, you *must* declare the anonymous function.

This is an example of a named function for integers that returns a function.

```
addx x == ((y:  Integer):  Integer +-> x + y)
```

                                                              Type: Void

We define **g** to be a function that adds 10 to its argument.

```
g := addx 10
```

   Compiling function addx with type
     PositiveInteger -> (Integer -> Integer)

                              theMap(...)

                                      Type: (Integer -> Integer)

Try it out.

```
g 3
```

                                  13

                                      Type: PositiveInteger

```
g(-4)
```

$$6$$

```
                                              Type:  PositiveInteger
```

An anonymous function cannot be recursive: since it does not have a name, you cannot even call it within itself! If you place an anonymous function inside a named function, the anonymous function must be declared.

6.18 Example: A Database

This example shows how you can use Axiom to organize a database of lineage data and then query the database for relationships.

The database is entered as "assertions" that are really pieces of a function definition.

```
children("albert") == ["albertJr","richard","diane"]
```

```
                                                         Type:  Void
```

Each piece $children(x) == y$ means "the children of x are y".

```
children("richard") == ["douglas","daniel","susan"]
```

```
                                                         Type:  Void
```

This family tree thus spans four generations.

```
children("douglas") == ["dougie","valerie"]
```

```
                                                         Type:  Void
```

Say "no one else has children."

```
children(x) == []
```

```
                                                         Type:  Void
```

We need some functions for computing lineage. Start with `childOf`.

```
childOf(x,y) == member?(x,children(y))
```

```
                                        Type:  Void
```

To find the `parentOf` someone, you have to scan the database of people applying `children`.

```
parentOf(x) ==
  for y in people repeat
    (if childOf(x,y) then return y)
  "unknown"
```

```
                                        Type:  Void
```

And a grandparent of x is just a parent of a parent of x.

```
grandParentOf(x) == parentOf parentOf x
```

```
                                        Type:  Void
```

The grandchildren of x are the people y such that x is a grandparent of y.

```
grandchildren(x) == [y for y in people | grandParentOf(y) = x]
```

```
                                        Type:  Void
```

Suppose you want to make a list of all great-grandparents. Well, a great-grandparent is a grandparent of a person who has children.

```
greatGrandParents == [x for x in people |
  reduce(\_or,
    [not empty? children(y) for y in grandchildren(x)],false)]
```

```
                                        Type:  Void
```

Define `descendants` to include the parent as well.

```
descendants(x) ==
  kids := children(x)
  null kids => [x]
  concat(x,reduce(concat,[descendants(y)
    for y in kids],[]))
```

```
                                        Type:  Void
```

Finally, we need a list of people. Since all people are descendants of "albert", let's say so.

```
people == descendants "albert"
```

<div align="right">

Type: Void

</div>

We have used "==" to define the database and some functions to query the database. But no computation is done until we ask for some information. Then, once and for all, the functions are analyzed and compiled to machine code for run-time efficiency. Notice that no types are given anywhere in this example. They are not needed.

Who are the grandchildren of "richard"?

```
grandchildren "richard"
```

```
Compiling function children with type String -> List String
Compiling function descendants with type String -> List String
Compiling body of rule people to compute value of type List String
Compiling function childOf with type (String,String) -> Boolean
Compiling function parentOf with type String -> String
Compiling function grandParentOf with type String -> String
Compiling function grandchildren with type String -> List String
```

<div align="center">

$\big[$"dougie","valerie"$\big]$

</div>

<div align="right">

Type: List String

</div>

Who are the great-grandparents?

```
greatGrandParents
```

```
Compiling body of rule greatGrandParents to compute value of
   type List String
```

<div align="center">

$\big[$"albert"$\big]$

</div>

<div align="right">

Type: List String

</div>

6.19 Example: A Famous Triangle

In this example we write some functions that display Pascal's triangle. It demonstrates the use of piece-wise definitions and some output operations you probably haven't seen before.

To make these output operations available, we have to *expose* the domain `OutputForm`. See 2.11 on page 168 for more information about exposing domains and packages.

```
)set expose add constructor OutputForm
```

```
   OutputForm is now explicitly exposed in frame G82322
```

Define the values along the first row and any column i.

```
pascal(1,i) == 1
```

```
                                                      Type:   Void
```

Define the values for when the row and column index i are equal. Repeating the argument name indicates that the two index values are equal.

```
pascal(n,n) == 1
```

```
                                                      Type:   Void
```

```
pascal(i,j | 1 < i and i < j) ==
   pascal(i-1,j-1)+pascal(i,j-1)
```

```
                                                      Type:   Void
```

Now that we have defined the coefficients in Pascal's triangle, let's write a couple of one-liners to display it.

First, define a function that gives the n-th row.

```
pascalRow(n) == [pascal(i,n) for i in 1..n]
```

```
                                                      Type:   Void
```

Next, we write the function **displayRow** to display the row, separating entries by blanks and centering.

```
displayRow(n) == output center blankSeparate pascalRow(n)
```

                                                          Type: Void

Here we have used three output operations. Operation **output** displays the
printable form of objects on the screen, **center** centers a printable form in the
width of the screen, and **blankSeparate** takes a list of nprintable forms and
inserts a blank between successive elements.

Look at the result.

```
for i in 1..7 repeat displayRow i
```

```
   Compiling function pascal with type (Integer,Integer) ->
      PositiveInteger
   Compiling function pascalRow with type PositiveInteger -> List
      PositiveInteger
   Compiling function displayRow with type PositiveInteger -> Void
```

```
                 1
                1 1
               1 2 1
              1 3 3 1
             1 4 6 4 1
            1 5 10 10 5 1
           1 6 15 20 15 6 1
```

                                                          Type: Void

Being purists, we find this less than satisfactory. Traditionally, elements of
Pascal's triangle are centered between the left and right elements on the line
above.

To fix this misalignment, we go back and redefine **pascalRow** to right adjust
the entries within the triangle within a width of four characters.

```
pascalRow(n) == [right(pascal(i,n),4) for i in 1..n]
```

```
   Compiled code for pascalRow has been cleared.
   Compiled code for displayRow has been cleared.
   1 old definition(s) deleted for function or rule pascalRow
```

                                                          Type: Void

Finally let's look at our purely reformatted triangle.

```
for i in 1..7 repeat displayRow i
```

```
   Compiling function pascalRow with type PositiveInteger -> List
      OutputForm
```

```
+++ |*1;pascalRow;1;G82322| redefined
   Compiling function displayRow with type PositiveInteger -> Void
```

```
+++ |*1;displayRow;1;G82322| redefined
                                 1
                              1     1
                           1     2     1
                        1     3     3     1
                     1     4     6     4     1
                  1     5    10    10     5     1
               1     6    15    20    15     6     1
```

```
                                                     Type:  Void
```

Unexpose `OutputForm` so we don't get unexpected results later.

```
)set expose drop constructor OutputForm
```

```
   OutputForm is now explicitly hidden in frame G82322
```

6.20 Example: Testing for Palindromes

In this section we define a function **pal?** that tests whether its argument is a *palindrome*, that is, something that reads the same backwards and forwards. For example, the string "Madam I'm Adam" is a palindrome (excluding blanks and punctuation) and so is the number 123454321. The definition works for any datatype that has n components that are accessed by the indices $1 \ldots n$.

Here is the definition for **pal?**. It is simply a call to an auxiliary function called **palAux?**. We are following the convention of ending a function's name with ? if the function returns a `Boolean` value.

```
pal?  s == palAux?(s,1,#s)
```

```
                                                     Type:  Void
```

Here is **palAux?**. It works by comparing elements that are equidistant from the start and end of the object.

```
palAux?(s,i,j) ==
  j > i =>
    (s.i = s.j) and palAux?(s,i+1,i-1)
  true
```

<div align="right">

Type: Void

</div>

Try **pal?** on some examples. First, a string.

```
pal?  "Oxford"
```

```
   Compiling function palAux? with type (String,Integer,Integer) ->
      Boolean
   Compiling function pal? with type String -> Boolean
```

<div align="center">

false

</div>

<div align="right">

Type: Boolean

</div>

A list of polynomials.

```
pal?  [4,a,x-1,0,x-1,a,4]
```

```
   Compiling function palAux? with type (List Polynomial Integer,
      Integer,Integer) -> Boolean
   Compiling function pal? with type List Polynomial Integer -> Boolean
```

<div align="center">

true

</div>

<div align="right">

Type: Boolean

</div>

A list of integers from the example in the last section.

```
pal?  [1,6,15,20,15,6,1]
```

```
   Compiling function palAux? with type (List PositiveInteger,Integer,
      Integer) -> Boolean
   Compiling function pal? with type List PositiveInteger -> Boolean
```

<div align="center">

true

</div>

<div align="right">Type: Boolean</div>

To use **pal?** on an integer, first convert it to a string.

```
pal?(1441::String)
```

<div align="center">true</div>

<div align="right">Type: Boolean</div>

Compute an infinite stream of decimal numbers, each of which is an obvious palindrome.

```
ones := [reduce(+,[10**j for j in 0..i]) for i in 1..]
```

$$[11, 111, 1111, 11111, 111111, 1111111,$$

$$11111111, 111111111, 1111111111, 11111111111, \ldots]$$

<div align="right">Type: Stream PositiveInteger</div>

```
)set streams calculate 9
```

How about their squares?

```
squares := [x**2 for x in ones]
```

$$[121, 12321, 1234321, 123454321, 12345654321, 1234567654321,$$

$$123456787654321, 12345678987654321, 1234567900987654321,$$

$$123456790120987654321, \ldots]$$

<div align="right">Type: Stream PositiveInteger</div>

Well, let's test them all.

```
[pal?(x::String) for x in squares]
```

$$[\text{true}, \text{true}, \text{true}, \text{true}, \text{true}, \text{true}, \text{true}, \text{true}, \text{true}, \text{true}, \ldots]$$

<div align="right">Type: Stream Boolean</div>

```
)set streams calculate 7
```

6.21   Rules and Pattern Matching

A common mathematical formula is

$$\log(x) + \log(y) = \log(xy) \quad \forall\, x \text{ and } y$$

The presence of "$\forall$" indicates that x and y can stand for arbitrary mathematical expressions in the above formula. You can use such mathematical formulas in Axiom to specify "rewrite rules". Rewrite rules are objects in Axiom that can be assigned to variables for later use, often for the purpose of simplification. Rewrite rules look like ordinary function definitions except that they are preceded by the reserved word *rule*. For example, a rewrite rule for the above formula is:

```
rule log(x) + log(y) == log(x * y)
```

Like function definitions, no action is taken when a rewrite rule is issued. Think of rewrite rules as functions that take one argument. When a rewrite rule $A = B$ is applied to an argument f, its meaning is: "rewrite every subexpression of f that *matches* A by B." The left-hand side of a rewrite rule is called a *pattern*; its right-side side is called its *substitution*.

Create a rewrite rule named **logrule**. The generated symbol beginning with a "%" is a place-holder for any other terms that might occur in the sum.

```
logrule := rule log(x) + log(y) == log(x * y)
```

$$\log\left(y\right) + \log\left(x\right) + \%C == \log\left(x\,y\right) + \%C$$

<div align="center">Type: RewriteRule(Integer,Integer,Expression Integer)</div>

Create an expression with logarithms.

```
f := log sin x + log x
```

$$\log\left(\sin\left(x\right)\right) + \log\left(x\right)$$

<div align="right">Type: Expression Integer</div>

Apply **logrule** to f.

```
logrule f
```

$$\log\left(x\,\sin\left(x\right)\right)$$

<div align="right">Type: Expression Integer</div>

The meaning of our example rewrite rule is: "for all expressions x and y, rewrite $log(x) + log(y)$ by $log(x * y)$." Patterns generally have both operation names (here, **log** and "**+**") and variables (here, x and y). By default, every operation name stands for itself. Thus **log** matches only "*log*" and not any other operation such as **sin**. On the other hand, variables do not stand for themselves. Rather, a variable denotes a *pattern variable* that is free to match any expression whatsoever.

When a rewrite rule is applied, a process called *pattern matching* goes to work by systematically scanning the subexpressions of the argument. When a subexpression is found that "matches" the pattern, the subexpression is replaced by the right-hand side of the rule. The details of what happens will be covered later.

The customary Axiom notation for patterns is actually a shorthand for a longer, more general notation. Pattern variables can be made explicit by using a percent "%" as the first character of the variable name. To say that a name stands for itself, you can prefix that name with a quote operator "'". Although the current Axiom parser does not let you quote an operation name, this more general notation gives you an alternate way of giving the same rewrite rule:

```
rule log(%x) + log(%y) == log(x * y)
```

This longer notation gives you patterns that the standard notation won't handle. For example, the rule

```
rule %f(c * 'x) ==  c*%f(x)
```

means "for all f and c, replace $f(y)$ by $c * f(x)$ when y is the product of c and the explicit variable x."

Thus the pattern can have several adornments on the names that appear there. Normally, all these adornments are dropped in the substitution on the right-hand side.

To summarize:

To enter a single rule in Axiom, use the following syntax:

$$\texttt{rule}\ \textit{leftHandSide}\ \texttt{==}\ \textit{rightHandSide}$$

The *leftHandSide* is a pattern to be matched and the *rightHandSide* is its substitution. The rule is an object of type `RewriteRule` that can be assigned to a variable and applied to expressions to transform them.

Rewrite rules can be collected into rulesets so that a set of rules can be applied at once. Here is another simplification rule for logarithms.

$$y \log(x) = \log(x^y) \quad \forall\, x \text{ and } y$$

If instead of giving a single rule following the reserved word *rule* you give a "pile" of rules, you create what is called a *ruleset*. Like rules, rulesets are objects in Axiom and can be assigned to variables. You will find it useful to group commonly used rules into input files, and read them in as needed.

Create a ruleset named *logrules*.

```
logrules := rule
  log(x) + log(y)  == log(x * y)
  y * log x        == log(x ** y)
```

$$\{\log(y) + \log(x) + \%B == \log(x\ y) + \%B, y\ \log(x) == \log(x^y)\}$$

<div align="center">

Type: Ruleset(Integer,Integer,Expression Integer)

</div>

Again, create an expression f containing logarithms.

```
f := a * log(sin x) - 2 * log x
```

$$a\ \log(\sin(x)) - 2\ \log(x)$$

<div align="center">

Type: Expression Integer

</div>

Apply the ruleset **logrules** to f.

```
logrules f
```

$$\log\left(\frac{\sin(x)^a}{x^2}\right)$$

<div align="center">

Type: Expression Integer

</div>

We have allowed pattern variables to match arbitrary expressions in the above examples. Often you want a variable only to match expressions satisfying some predicate. For example, we may want to apply the transformation

$$y \log(x) = \log(x^y)$$

only when y is an integer.

The way to restrict a pattern variable y by a predicate $f(y)$ is by using a vertical bar "|", which means "such that," in much the same way it is used in function definitions. You do this only once, but at the earliest (meaning deepest and leftmost) part of the pattern.

This restricts the logarithmic rule to create integer exponents only.

```
logrules2 := rule
  log(x) + log(y)           == log(x * y)
  (y | integer? y) * log x == log(x ** y)
```

$$\{\log\left(y\right) + \log\left(x\right) + \%D == \log\left(x\ y\right) + \%D, y\ \log\left(x\right) == \log\left(x^{y}\right)\}$$

<div align="center">Type: Ruleset(Integer,Integer,Expression Integer)</div>

Compare this with the result of applying the previous set of rules.

```
f
```

$$a\ \log\left(\sin\left(x\right)\right) - 2\ \log\left(x\right)$$

<div align="center">Type: Expression Integer</div>

```
logrules2 f
```

$$a\ \log\left(\sin\left(x\right)\right) + \log\left(\frac{1}{x^2}\right)$$

<div align="center">Type: Expression Integer</div>

You should be aware that you might need to apply a function like `integer` within your predicate expression to actually apply the test function.

Here we use `integer` because n has type **Expression Integer** but **even?** is an operation defined on integers.

```
evenRule := rule cos(x)**(n | integer? n and even? integer
n)==(1-sin(x)**2)**(n/2)
```

$$\cos\left(x\right)^{n} == \left(-\sin\left(x\right)^{2} + 1\right)^{\frac{n}{2}}$$

<div align="center">Type: RewriteRule(Integer,Integer,Expression Integer)</div>

Here is the application of the rule.

```
evenRule( cos(x)**2 )
```

$$-\sin\left(x\right)^{2} + 1$$

<div align="center">Type: Expression Integer</div>

This is an example of some of the usual identities involving products of sines and cosines.

```
sinCosProducts == rule
  sin(x) * sin(y) == (cos(x-y) - cos(x + y))/2
  cos(x) * cos(y) == (cos(x-y) + cos(x+y))/2
  sin(x) * cos(y) == (sin(x-y) + sin(x + y))/2
```

<div align="right">Type: Void</div>

```
g := sin(a)*sin(b) + cos(b)*cos(a) + sin(2*a)*cos(2*a)
```

$$\sin\left(a\right)\,\sin\left(b\right) + \cos\left(2\ a\right)\,\sin\left(2\ a\right) + \cos\left(a\right)\,\cos\left(b\right)$$

<div align="right">Type: Expression Integer</div>

```
sinCosProducts g
```

> Compiling body of rule sinCosProducts to compute value of type
>   Ruleset(Integer,Integer,Expression Integer)

$$\frac{\sin\left(4\ a\right) + 2\,\cos\left(b - a\right)}{2}$$

<div align="right">Type: Expression Integer</div>

Another qualification you will often want to use is to allow a pattern to match an identity element. Using the pattern $x + y$, for example, neither x nor y matches the expression 0. Similarly, if a pattern contains a product $x * y$ or an exponentiation $x * *y$, then neither x or y matches 1.

If identical elements were matched, pattern matching would generally loop. Here is an expansion rule for exponentials.

```
exprule := rule exp(a + b) == exp(a) * exp(b)
```

$$e^{(b+a)} == e^a\ e^b$$

<div align="center">Type: RewriteRule(Integer,Integer,Expression Integer)</div>

This rule would cause infinite rewriting on this if either a or b were allowed to match 0.

```
exprule exp x
```

$$e^x$$

<div align="right">Type: Expression Integer</div>

There are occasions when you do want a pattern variable in a sum or product to match 0 or 1. If so, prefix its name with a "?" whenever it appears in a left-hand side of a rule. For example, consider the following rule for the exponential integral:

$$\int \left(\frac{y + e^x}{x} \right) dx = \int \frac{y}{x} dx + \text{Ei}(x) \quad \forall \, x \text{ and } y$$

This rule is valid for $y = 0$. One solution is to create a `Ruleset` with two rules, one with and one without y. A better solution is to use an "optional" pattern variable.

Define rule `eirule` with a pattern variable ?y to indicate that an expression may or may not occur.

```
eirule := rule integral((?y + exp x)/x,x) == integral(y/x,x) + Ei
x
```

$$\int^x \frac{e^{\%M} + y}{\%M} \, d\%M == {}'integral \left(\frac{y}{x}, x \right) + {}'Ei \, (x)$$

<div align="center">Type: RewriteRule(Integer,Integer,Expression Integer)</div>

Apply rule `eirule` to an integral without this term.

```
eirule integral(exp u/u, u)
```

$$Ei \, (u)$$

<div align="right">Type: Expression Integer</div>

Apply rule `eirule` to an integral with this term.

```
eirule integral(sin u + exp u/u, u)
```

$$\int^u \sin \left(\%M \right) \, d\%M + Ei \, (u)$$

<div align="right">Type: Expression Integer</div>

Here is one final adornment you will find useful. When matching a pattern of the form $x+y$ to an expression containing a long sum of the form $a+\ldots+b$, there is no way to predict in advance which subset of the sum matches x and which matches y. Aside from efficiency, this is generally unimportant since the rule holds for any possible combination of matches for x and y. In some situations, however, you may want to say which pattern variable is a sum (or product) of several terms, and which should match only a single term. To do this, put a prefix colon ":" before the pattern variable that you want to match multiple terms.

The remaining rules involve operators u and v.

```
u := operator 'u
```

$$u$$

<div align="right">

Type: BasicOperator
</div>

These definitions tell Axiom that u and v are formal operators to be used in expressions.

```
v := operator 'v
```

$$v$$

<div align="right">

Type: BasicOperator
</div>

First define `myRule` with no restrictions on the pattern variables x and y.

```
myRule := rule u(x + y) == u x + v y
```

$$u\,(y+x)=='v\,(y)+'u\,(x)$$

<div align="center">

Type: RewriteRule(Integer,Integer,Expression Integer)
</div>

Apply `myRule` to an expression.

```
myRule u(a + b + c + d)
```

$$v\,(d+c+b)+u\,(a)$$

<div align="right">

Type: Expression Integer
</div>

Define `myOtherRule` to match several terms so that the rule gets applied recursively.

```
myOtherRule := rule u(:x + y) == u x + v y
```

$$u\,(y+x) == \ 'v\,(y) + \ 'u\,(x)$$

> Type: RewriteRule(Integer,Integer,Expression Integer)

Apply `myOtherRule` to the same expression.

```
myOtherRule u(a + b + c + d)
```

$$v\,(c) + v\,(b) + v\,(a) + u\,(d)$$

> Type: Expression Integer

Summary of pattern variable adornments:

`(x | predicate?(x))`	means that the substutution s for x must satisfy `predicate(s) = true`.
`?x`	means that x can match an identity element (0 or 1).
`:x`	means that x can match several terms in a sum.

Here are some final remarks on pattern matching. Pattern matching provides a very useful paradigm for solving certain classes of problems, namely, those that involve transformations of one form to another and back. However, it is important to recognize its limitations.

First, pattern matching slows down as the number of rules you have to apply increases. Thus it is good practice to organize the sets of rules you use optimally so that irrelevant rules are never included.

Second, careless use of pattern matching can lead to wrong answers. You should avoid using pattern matching to handle hidden algebraic relationships that can go undetected by other programs. As a simple example, a symbol such as "J" can easily be used to represent the square root of -1 or some other important algebraic quantity. Many algorithms branch on whether an expression is zero or not, then divide by that expression if it is not. If you fail to simplify an expression involving powers of J to -1, algorithms may incorrectly assume an expression is non-zero, take a wrong branch, and produce a meaningless result.

Pattern matching should also not be used as a substitute for a domain. In Axiom, objects of one domain are transformed to objects of other domains using well-defined **coerce** operations. Pattern matching should be used on objects that are all the same type. Thus if your application can be handled by type `Expression` in Axiom and you think you need pattern matching, consider this choice carefully. You may well be better served by extending an existing domain or by building a new domain of objects for your application.

Chapter 7

Graphics
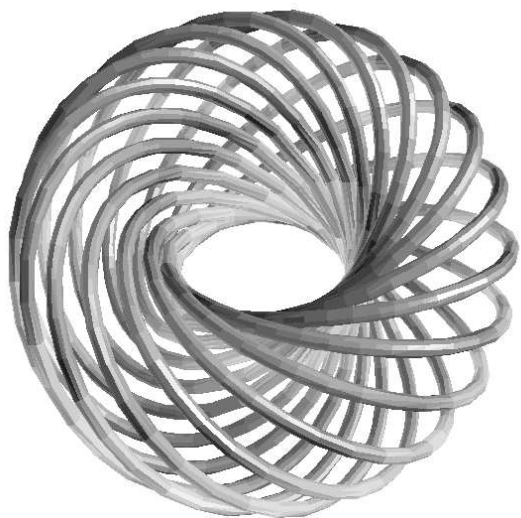
Figure 7.1: Torus knot of type (15,17).

This chapter shows how to use the Axiom graphics facilities under the X Window System. Axiom has two-dimensional and three-dimensional drawing and rendering packages that allow the drawing, coloring, transforming, mapping, clipping, and combining of graphic output from Axiom computations. This facility is particularly useful for investigating problems in areas such as topology. The graphics package is capable of plotting functions of one or more variables or plotting parametric surfaces and curves. Various coordinate systems are also available, such as polar and spherical.

A graph is displayed in a viewport window and it has a control-panel that uses interactive mouse commands. PostScript and other output forms are available so that Axiom images can be printed or used by other programs.

7.1   Two-Dimensional Graphics

The Axiom two-dimensional graphics package provides the ability to display

- curves defined by functions of a single real variable

- curves defined by parametric equations

- implicit non-singular curves defined by polynomial equations

- planar graphs generated from lists of point components.

These graphs can be modified by specifying various options, such as calculating points in the polar coordinate system or changing the size of the graph viewport window.

7.1.1   Plotting Two-Dimensional Functions of One Variable

The first kind of two-dimensional graph is that of a curve defined by a function $y = f(x)$ over a finite interval of the x axis.

The general format for drawing a function defined by a formula $f(x)$ is:

$$\texttt{draw(f(x), x = a..b, } options\texttt{)}$$

where $a..b$ defines the range of x, and where *options* prescribes zero or more options as described in 7.1.4 on page 301. An example of an option is $curveColor == brightred()$. An alternative format involving functions f and g is also available.

A simple way to plot a function is to use a formula. The first argument is the formula. For the second argument, write the name of the independent variable (here, x), followed by an "=", and the range of values.

Display this formula over the range $0 \le x \le 6$. Axiom converts your formula to a compiled function so that the results can be computed quickly and efficiently.

```
draw(sin(tan(x)) - tan(sin(x)),x = 0..6)
```

Once again the formula is converted to a compiled function before any points were computed. If you want to graph the same function on several intervals, it is a good idea to define the function first so that the function has to be compiled only once.

This time we first define the function.

```
f(x) == (x-1)*(x-2)*(x-3)
```

Type: Void

To draw the function, the first argument is its name and the second is just the range with no independent variable.

```
draw(f, 0..4)
```

7.1.2   Plotting Two-Dimensional Parametric Plane Curves

The second kind of two-dimensional graph is that of curves produced by parametric equations. Let $x = f(t)$ and $y = g(t)$ be formulas or two functions f and g as the parameter t ranges over an interval $[a, b]$. The function **curve** takes the two functions f and g as its parameters.

The general format for drawing a two-dimensional plane curve defined by parametric formulas $x = f(t)$ and $y = g(t)$ is:

```
draw(curve(f(t), g(t)), t = a..b, options)
```

where $a..b$ defines the range of the independent variable t, and where *options* prescribes zero or more options as described in 7.2.4 on page 322. An example of an option is $curveColor == brightred()$.

Here's an example:

Define a parametric curve using a range involving $\%pi$, Axiom's way of saying π. For parametric curves, Axiom compiles two functions, one for each of the functions f and g.

```
draw(curve(sin(t)*sin(2*t)*sin(3*t), sin(4*t)*sin(5*t)*sin(6*t)),
t = 0..2*%pi)
```

The title may be an arbitrary string and is an optional argument to the **draw** command.

```
draw(curve(cos(t), sin(t)), t = 0..2*%pi)
```

If you plan on plotting $x = f(t)$, $y = g(t)$ as t ranges over several intervals, you may want to define functions f and g first, so that they need not be recompiled every time you create a new graph. Here's an example:

As before, you can first define the functions you wish to draw.

```
f(t:DFLOAT):DFLOAT == sin(3*t/4)
```

   Function declaration f : DoubleFloat -> DoubleFloat has been
      added to workspace.

                                                          Type: Void

Axiom compiles them to map `DoubleFloat` values to `DoubleFloat` values.

```
g(t:DFLOAT):DFLOAT == sin(t)
```

   Function declaration f : DoubleFloat -> DoubleFloat has been added
      to workspace.

                                                          Type: Void

Give to **curve** the names of the functions, then write the range without the name of the independent variable.

```
draw(curve(f,g),0..%pi)
```

Here is another look at the same curve but over a different range. Notice that f and g are not recompiled. Also note that Axiom provides a default title based on the first function specified in **curve**.

```
draw(curve(f,g),-4*%pi..4*%pi)
```

7.1.3   Plotting Plane Algebraic Curves

A third kind of two-dimensional graph is a non-singular "solution curve" in a rectangular region of the plane. A solution curve is a curve defined by a polynomial equation $p(x, y) = 0$. Non-singular means that the curve is "smooth" in that it does not cross itself or come to a point (cusp). Algebraically, this means that for any point (x, y) on the curve, that is, a point such that $p(x, y) = 0$, the partial derivatives $\frac{\partial p}{\partial x}(x, y)$ and $\frac{\partial p}{\partial y}(x, y)$ are not both zero.

The general format for drawing a non-singular solution curve given by a polynomial of the form $p(x, y) = 0$ is:

```
draw(p(x,y) = 0, x, y, range == [a..b, c..d], options)
```

where the second and third arguments name the first and second independent variables of p. A `range` option is always given to designate a bounding rectangular region of the plane $a \leq x \leq b, c \leq y \leq d$. Zero or more additional options as described in 7.1.4 on page 301 may be given.

We require that the polynomial has rational or integral coefficients. Here is an algebraic curve example ("Cartesian ovals"):

```
p := ((x**2 + y**2 + 1) - 8*x)**2 - (8*(x**2 + y**2 + 1)-4*x-1)
```

```
$$
{y \sp 4}+{{\left( {2 \  {x \sp 2}} -{{16} \  x} -6
\right)}
\  {y \sp 2}}+{x \sp 4} -{{16} \  {x \sp 3}}+{{58} \  {x \sp 2}} -{{12} \  x}
-6
$$
```

```
                                  Type:  Polynomial Integer
```

The first argument is always expressed as an equation of the form $p = 0$ where p is a polynomial.

```
draw(p = 0, x, y, range == [-1..11, -7..7])
```

7.1.4 Two-Dimensional Options

The **draw** commands take an optional list of options, such as `title` shown above. Each option is given by the syntax: *name* `==` *value*. Here is a list of the available options in the order that they are described below.

adaptive	clip	unit
clip	curveColor	range
toScale	pointColor	coordinates

The *adaptive* option turns adaptive plotting on or off. Adaptive plotting uses an algorithm that traverses a graph and computes more points for those parts of the graph with high curvature. The higher the curvature of a region is, the more points the algorithm computes.

The `adaptive` option is normally on. Here we turn it off.

```
draw(sin(1/x),x=-2*%pi..2*%pi, adaptive == false)
```

The `clip` option turns clipping on or off. If on, large values are cut off according to **clipPointsDefault**.

```
draw(tan(x),x=-2*%pi..2*%pi, clip == true)
```

Option `toScale` does plotting to scale if `true` or uses the entire viewport if `false`. The default can be determined using **drawToScale**.

```
draw(sin(x),x=-%pi..%pi, toScale == true, unit == [1.0,1.0])
```

Option `clip` with a range sets point clipping of a graph within the ranges specified in the list $[xrange, yrange]$. If only one range is specified, clipping applies to the y-axis.

```
draw(sec(x),x=-2*%pi..2*%pi, clip == [-2*%pi..2*%pi,-%pi..%pi],
unit == [1.0,1.0])
```

Option `curveColor` sets the color of the graph curves or lines to be the indicated palette color (see 7.1.5 on page 303 and 7.1.6 on page 304).

```
draw(sin(x),x=-%pi..%pi, curveColor == bright red())
```

Option `pointColor` sets the color of the graph points to the indicated palette color (see 7.1.5 on page 303 and 7.1.6 on page 304).

```
draw(sin(x),x=-%pi..%pi, pointColor == pastel yellow())
```

Option `unit` sets the intervals at which the axis units are plotted according to the indicated steps $[x \text{ interval}, y \text{ interval}]$.

```
draw(curve(9*sin(3*t/4),8*sin(t)), t = -4*%pi..4*%pi, unit ==
[2.0,1.0])
```

Option `range` sets the range of variables in a graph to be within the ranges for solving plane algebraic curve plots.

```
draw(y**2 + y - (x**3 - x) = 0, x, y, range == [-2..2,-2..1],
unit==[1.0,1.0])
```

A second example of a solution plot.

```
draw(x**2 + y**2 = 1, x, y, range == [-3/2..3/2,-3/2..3/2],
unit==[0.5,0.5])
```

Option *coordinates* indicates the coordinate system in which the graph is plotted. The default is to use the Cartesian coordinate system. For more details, see 7.2.7 on page 331 or `CoordinateSystems`.

```
draw(curve(sin(5*t),t),t=0..2*%pi, coordinates == polar)
```

7.1.5   Color

The domain `Color` provides operations for manipulating colors in two-dimensional graphs. Colors are objects of `Color`. Each color has a *hue* and a *weight*. Hues are represented by integers that range from 1 to the **numberOfHues()**, normally 27. Weights are floats and have the value 1.0 by default.

color (*integer*)
> creates a color of hue *integer* and weight 1.0.

hue (*color*)
> returns the hue of *color* as an integer.

red ()
> **blue**(), **green**(), and **yellow**() create colors of that hue with weight 1.0.

color$_1$ + *color*$_2$ returns the color that results from additively combining the indicated *color*$_1$ and *color*$_2$. Color addition is not commutative: changing the order of the arguments produces different results.

integer ∗ *color* changes the weight of *color* by *integer* without affecting its hue. For example, $red() + 3 * yellow()$ produces a color closer to yellow than to red. Color multiplication is not associative: changing the order of grouping produces different results.

These functions can be used to change the point and curve colors for two- and three-dimensional graphs. Use the `pointColor` option for points.

```
draw(x**2,x=-1..1,pointColor == green())
```

Use the `curveColor` option for curves.

```
draw(x**2,x=-1..1,curveColor == color(13) + 2*blue())
```

7.1.6   Palette

Domain `Palette` is the domain of shades of colors: **dark**, **dim**, **bright**, **pastel**, and **light**, designated by the integers 1 through 5, respectively.

Colors are normally "bright."

```
shade red()
```

$$3$$

                                            Type: PositiveInteger

To change the shade of a color, apply the name of a shade to it.

```
myFavoriteColor := dark blue()
```

[Hue: 22Weight: 1.0] from the *Dark*palette

                                                    Type: Palette

The expression *shade*(*color*) returns the value of a shade of *color*.

```
shade myFavoriteColor
```

$$1$$

                                            Type: PositiveInteger

The expression *hue*(*color*) returns its hue.

```
hue myFavoriteColor
```

Hue: 22Weight: 1.0

                                                    Type: Color

Palettes can be used in specifying colors in two-dimensional graphs.

```
draw(x**2,x=-1..1,curveColor == dark blue())
```

Figure 7.2: Two-dimensional control-panel.

7.1.7   Two-Dimensional Control-Panel

Once you have created a viewport, move your mouse to the viewport and click with your left mouse button to display a control-panel. The panel is displayed on the side of the viewport closest to where you clicked. Each of the buttons which toggle on and off show the current state of the graph.

Transformations

Object transformations are executed from the control-panel by mouse-activated potentiometer windows.

Scale: To scale a graph, click on a mouse button within the **Scale** window in the upper left corner of the control-panel. The axes along which the scaling is to occur are indicated by setting the toggles above the arrow. With `X On` and `Y On` appearing, both axes are selected and scaling is uniform. If either is not selected, for example, if `X Off` appears, scaling is non-uniform.

Translate: To translate a graph, click the mouse in the **Translate** window in the direction you wish the graph to move. This window is located in the upper right corner of the control-panel. Along the top of the **Translate** window are two buttons for selecting the direction of translation. Translation along both coordinate axes results when `X On` and `Y On` appear or along one axis when one is on, for example, `X On` and `Y Off` appear.

Messages

The window directly below the transformation potentiometer windows is used to display system messages relating to the viewport and the control-panel. The following format is displayed:

[scaleX, scaleY] >graph< [translateX, translateY]

The two values to the left show the scale factor along the `X` and `Y` coordinate axes. The two values to the right show the distance of translation from the center in the `X` and `Y` directions. The number in the center shows which graph in the viewport this data pertains to. When multiple graphs exist in the same viewport, the graph must be selected (see "Multiple Graphs," below) in order for its transformation data to be shown, otherwise the number is 1.

Multiple Graphs

The **Graphs** window contains buttons that allow the placement of two-dimensional graphs into one of nine available slots in any other two-dimensional viewport. In the center of the window are numeral buttons from one to nine that show whether a graph is displayed in the viewport. Below each number button is a button showing whether a graph that is present is selected for application of some transformation. When the caret symbol is displayed, then the graph in that slot will be manipulated. Initially, the graph for which the viewport is created occupies the first slot, is displayed, and is selected.

Clear: The **Clear** button deselects every viewport graph slot. A graph slot is reselected by selecting the button below its number.

Query: The **Query** button is used to display the scale and translate data for the indicated graph. When this button is selected the message "Click on the graph to query" appears. Select a slot number button from the **Graphs** window. The scaling factor and translation offset of the graph are then displayed in the message window.

Pick: The **Pick** button is used to select a graph to be placed or dropped into the indicated viewport. When this button is selected, the message "Click on the graph to pick" appears. Click on the slot with the graph number of the desired graph. The graph information is held waiting for you to execute a **Drop** in some other graph.

Drop: Once a graph has been picked up using the **Pick** button, the **Drop** button places it into a new viewport slot. The message "Click on the graph to drop" appears in the message window when the **Drop** button is selected. By selecting one of the slot number buttons in the **Graphs** window, the graph currently being held is dropped into this slot and displayed.

Buttons

Axes turns the coordinate axes on or off.

Units turns the units along the x and y axis on or off.

Box encloses the area of the viewport graph in a bounding box, or removes the box if already enclosed.

Pts turns on or off the display of points.

Lines turns on or off the display of lines connecting points.

PS writes the current viewport contents to a file **axiom2D.ps** or to a name specified in the user's **.Xdefaults** file. The file is placed in the directory from which Axiom or the **viewAlone** program was invoked.

Reset resets the object transformation characteristics and attributes back to their initial states.

Hide makes the control-panel disappear.

Quit queries whether the current viewport session should be terminated.

7.1.8 Operations for Two-Dimensional Graphics

Here is a summary of useful Axiom operations for two-dimensional graphics. Each operation name is followed by a list of arguments. Each argument is written as a variable informally named according to the type of the argument (for example, *integer*). If appropriate, a default value for an argument is given in parentheses immediately following the name.

adaptive ([*boolean*(`true`)])

    sets or indicates whether graphs are plotted according to the adaptive refinement algorithm.

axesColorDefault ([*color*(`dark blue()`)])

    sets or indicates the default color of the axes in a two-dimensional graph viewport.

clipPointsDefault ([*boolean*(`false`)])

    sets or indicates whether point clipping is to be applied as the default for graph plots.

drawToScale ([*boolean*(`false`)])

    sets or indicates whether the plot of a graph is "to scale" or uses the entire viewport space as the default.

lineColorDefault ([*color*(`pastel yellow()`)])

    sets or indicates the default color of the lines or curves in a two-dimensional graph viewport.

maxPoints ([*integer*(`500`)])

    sets or indicates the default maximum number of possible points to be used when constructing a two-dimensional graph.

minPoints ([*integer*(`21`)])

    sets or indicates the default minimum number of possible points to be used when constructing a two-dimensional graph.

pointColorDefault ([*color*(`bright red()`)])

    sets or indicates the default color of the points in a two-dimensional graph viewport.

pointSizeDefault ([*integer*(5)])

>   sets or indicates the default size of the dot used to plot points in a two-dimensional graph.

screenResolution ([*integer*(600)])

>   sets or indicates the default screen resolution constant used in setting the computation limit of adaptively generated curve plots.

unitsColorDefault ([*color*(`dim green()`)])

>   sets or indicates the default color of the unit labels in a two-dimensional graph viewport.

viewDefaults ()

>   resets the default settings for the following attributes: point color, line color, axes color, units color, point size, viewport upper left-hand corner position, and the viewport size.

viewPosDefault ([*list*(`[100,100]`)])

>   sets or indicates the default position of the upper left-hand corner of a two-dimensional viewport, relative to the display root window. The upper left-hand corner of the display is considered to be at the (0, 0) position.

viewSizeDefault ([*list*(`[200,200]`)])

>   sets or indicates the default size in which two dimensional viewport windows are shown. It is defined by a width and then a height.

viewWriteAvailable ([*list*(`["pixmap","bitmap", "postscript", "image"]`)])

>   indicates the possible file types that can be created with the **write** function.

viewWriteDefault ([*list*(`[]`)])

>   sets or indicates the default types of files, in addition to the **data** file, that are created when a **write** function is executed on a viewport.

units (*viewport, integer*(1), *string*(`"off"`))

>   turns the units on or off for the graph with index *integer*.

axes (*viewport, integer*(1), *string*(`"on"`))

>   turns the axes on or off for the graph with index *integer*.

close (*viewport*)

>   closes *viewport*.

connect (*viewport, integer*(1), *string*(`"on"`))

>   declares whether lines connecting the points are displayed or not.

controlPanel (*viewport, string*(`"off"`))

>   declares whether the two-dimensional control-panel is automatically displayed or not.

graphs (*viewport*)

    returns a list describing the state of each graph. If the graph state is not being used this is shown by `"undefined"`, otherwise a description of the graph's contents is shown.

graphStates (*viewport*)

    displays a list of all the graph states available for *viewport*, giving the values for every property.

key (*viewport*)

    returns the process ID number for *viewport*.

move (*viewport, integer$_x$(viewPosDefault), integer$_y$(viewPosDefault)*)

    moves *viewport* on the screen so that the upper left-hand corner of *viewport* is at the position *(x,y)*.

options (*viewport*)

    returns a list of all the `DrawOption`s used by *viewport*.

points (*viewport, integer*(`1`)*, string*(`"on"`))

    specifies whether the graph points for graph *integer* are to be displayed or not.

region (*viewport, integer*(`1`)*, string*(`"off"`))

    declares whether graph *integer* is or is not to be displayed with a bounding rectangle.

reset (*viewport*)

    resets all the properties of *viewport*.

resize (*viewport, integer$_{width}$,integer$_{height}$*)

    resizes *viewport* with a new *width* and *height*.

scale (*viewport, integer$_n$*(`1`)*, integer$_x$*(`0.9`)*, integer$_y$*(`0.9`))

    scales values for the x and y coordinates of graph n.

show (*viewport, integer$_n$*(`1`)*, string*(`"on"`))

    indicates if graph n is shown or not.

title (*viewport, string*(`"Axiom 2D"`))

    designates the title for *viewport*.

translate (*viewport, integer$_n$*(`1`)*, float$_x$*(`0.0`)*, float$_y$*(`0.0`))

    causes graph n to be moved x and y units in the respective directions.

write (*viewport, string$_{directory}$, [strings]*)

    if no third argument is given, writes the **data** file onto the directory with extension **data**. The third argument can be a single string or a list of strings with some or all the entries `"pixmap"`, `"bitmap"`, `"postscript"`, and `"image"`.

7.1.9   Addendum: Building Two-Dimensional Graphs

In this section we demonstrate how to create two-dimensional graphs from lists of points and give an example showing how to read the lists of points from a file.

Creating a Two-Dimensional Viewport from a List of Points

Axiom creates lists of points in a two-dimensional viewport by utilizing the `GraphImage` and `TwoDimensionalViewport` domains. In this example, the **makeGraphImage** function takes a list of lists of points parameter, a list of colors for each point in the graph, a list of colors for each line in the graph, and a list of sizes for each point in the graph.

The following expressions create a list of lists of points which will be read by Axiom and made into a two-dimensional viewport.

```
p1 := point [1,1]$(Point DFLOAT)
```

$$[1.0, 1.0]$$

                                        Type: Point DoubleFloat

```
p2 := point [0,1]$(Point DFLOAT)
```

$$[0.0, 1.0]$$

                                        Type: Point DoubleFloat

```
p3 := point [0,0]$(Point DFLOAT)
```

$$[0.0, 0.0]$$

                                        Type: Point DoubleFloat

```
p4 := point [1,0]$(Point DFLOAT)
```

$$[1.0, 0.0]$$

                                        Type: Point DoubleFloat

```
p5 := point [1,.5]$(Point DFLOAT)
```

$$[1.0, 0.5]$$

<div align="right">Type: Point DoubleFloat</div>

```
p6 := point [.5,0]$(Point DFLOAT)
```

$$[0.5, 0.0]$$

<div align="right">Type: Point DoubleFloat</div>

```
p7 := point [0,0.5]$(Point DFLOAT)
```

$$[0.0, 0.5]$$

<div align="right">Type: Point DoubleFloat</div>

```
p8 := point [.5,1]$(Point DFLOAT)
```

$$[0.5, 1.0]$$

<div align="right">Type: Point DoubleFloat</div>

```
p9 := point [.25,.25]$(Point DFLOAT)
```

$$[0.25, 0.25]$$

<div align="right">Type: Point DoubleFloat</div>

```
p10 := point [.25,.75]$(Point DFLOAT)
```

$$[0.25, 0.75]$$

<div align="right">Type: Point DoubleFloat</div>

```
p11 := point [.75,.75]$(Point DFLOAT)
```

$$[0.75, 0.75]$$

<div align="right">Type: Point DoubleFloat</div>

```
p12 := point [.75,.25]$(Point DFLOAT)
```

$$[0.75, 0.25]$$

<div align="right">Type: Point DoubleFloat</div>

Finally, here is the list.

```
llp := [ [p1,p2], [p2,p3], [p3,p4], [p4,p1], [p5,p6], [p6,p7],
[p7,p8], [p8,p5], [p9,p10], [p10,p11], [p11,p12], [p12,p9] ]
```

$[[[1.0, 1.0], [0.0, 1.0]], [[0.0, 1.0], [0.0, 0.0]], [[0.0, 0.0], [1.0, 0.0]], [[1.0, 0.0], [1.0, 1.0]], [[1.0, 0.5], [0.5, 0.0]], [[0.5,$

<div align="right">Type: List List Point DoubleFloat</div>

Now we set the point sizes for all components of the graph.

```
size1 := 6::PositiveInteger
```

$$6$$

<div align="right">Type: PositiveInteger</div>

```
size2 := 8::PositiveInteger
```

$$8$$

<div align="right">Type: PositiveInteger</div>

```
size3 := 10::PositiveInteger
```

```
lsize := [size1, size1, size1, size1, size2, size2, size2, size2,
size3, size3, size3, size3]
```

$$[6, 6, 6, 6, 8, 8, 8, 8, size3, size3, size3, size3]$$

<div align="right">Type: List Polynomial Integer</div>

Here are the colors for the points.

```
pc1 := pastel red()
```

[Hue: 1Weight: 1.0] from the *Pastel*palette

<div align="right">Type: Palette</div>

```
pc2 := dim green()
```

[Hue: 14Weight: 1.0] from the *Dim*palette

<div align="right">Type: Palette</div>

```
pc3 := pastel yellow()
```

[Hue: 11Weight: 1.0] from the *Pastel*palette

<div align="right">Type: Palette</div>

```
lpc := [pc1, pc1, pc1, pc1, pc2, pc2, pc2, pc2, pc3, pc3, pc3,
pc3]
```

[[Hue: 1Weight: 1.0] from the *Pastel*palette, [Hue: 1Weight: 1.0] from the *Pastel*palette, [Hue: 1Weight: 1.0] from t

<div align="right">Type: List Palette</div>

Here are the colors for the lines.

```
lc := [pastel blue(), light yellow(), dim green(), bright red(),
light green(), dim yellow(), bright blue(), dark red(), pastel
red(), light blue(), dim green(), light yellow()]
```

[[Hue: 22Weight: 1.0] from the *Pastel*palette, [Hue: 11Weight: 1.0] from the *Light*palette, [Hue: 14Weight: 1.0] from

<div align="right">Type: List Palette</div>

Now the `GraphImage` is created according to the component specifications indicated above.

```
g := makeGraphImage(llp,lpc,lc,lsize)$GRIMAGE
```

The **makeViewport2D** function now creates a `TwoDimensionalViewport` for this graph according to the list of options specified within the brackets.

```
makeViewport2D(g,[title("Lines")])$VIEW2D
```

This example demonstrates the use of the `GraphImage` functions **component** and **appendPoint** in adding points to an empty `GraphImage`.

```
)clear all
```

```
g := graphImage()$GRIMAGE
```

Graph with 0point lists

Type: GraphImage

```
p1 := point [0,0]$(Point DFLOAT)
```

$$[0.0, 0.0]$$

Type: Point DoubleFloat

```
p2 := point [.25,.25]$(Point DFLOAT)
```

$$[0.25, 0.25]$$

Type: Point DoubleFloat

```
p3 := point [.5,.5]$(Point DFLOAT)
```

$$[0.5, 0.5]$$

Type: Point DoubleFloat

```
p4 := point [.75,.75]$(Point DFLOAT)
```

$$[0.75, 0.75]$$

Type: Point DoubleFloat

```
p5 := point [1,1]$(Point DFLOAT)
```

$$[1.0, 1.0]$$

<div align="right">Type: Point DoubleFloat</div>

```
component(g,p1)$GRIMAGE
```

<div align="right">Type: Void</div>

```
component(g,p2)$GRIMAGE
```

<div align="right">Type: Void</div>

```
appendPoint(g,p3)$GRIMAGE
```

<div align="right">Type: Void</div>

```
appendPoint(g,p4)$GRIMAGE
```

<div align="right">Type: Void</div>

```
appendPoint(g,p5)$GRIMAGE
```

<div align="right">Type: Void</div>

```
g1 := makeGraphImage(g)$GRIMAGE
```

Here is the graph.

```
makeViewport2D(g1,[title("Graph Points")])$VIEW2D
```

A list of points can also be made into a `GraphImage` by using the operation
coerce. It is equivalent to adding each point to $g2$ using **component**.

```
g2 := coerce([ [p1],[p2],[p3],[p4],[p5] ])$GRIMAGE
```

Now, create an empty `TwoDimensionalViewport`.

```
v := viewport2D()$VIEW2D
```

```
options(v,[title("Just Points")])$VIEW2D
```

Place the graph into the viewport.

```
putGraph(v,g2,1)$VIEW2D
```

Take a look.

```
makeViewport2D(v)$VIEW2D
```

Creating a Two-Dimensional Viewport of a List of Points from a File

The following three functions read a list of points from a file and then draw the
points and the connecting lines. The points are stored in the file in readable form
as floating point numbers (specifically, `DoubleFloat` values) as an alternating
stream of x- and y-values. For example,

```
0.0 0.0     1.0 1.0     2.0 4.0
3.0 9.0     4.0 16.0    5.0 25.0
```

```
drawPoints(lp:List Point DoubleFloat):VIEW2D ==
  g := graphImage()$GRIMAGE
  for p in lp repeat
    component(g,p,pointColorDefault(),lineColorDefault(),
      pointSizeDefault())
  gi := makeGraphImage(g)$GRIMAGE
  makeViewport2D(gi,[title("Points")])$VIEW2D
```

```
drawLines(lp:List Point DoubleFloat):VIEW2D ==
  g := graphImage()$GRIMAGE
  component(g, lp, pointColorDefault(), lineColorDefault(),
    pointSizeDefault())$GRIMAGE
  gi := makeGraphImage(g)$GRIMAGE
  makeViewport2D(gi,[title("Points")])$VIEW2D
```

```
plotData2D(name, title) ==
  f:File(DFLOAT) := open(name,"input")
  lp:LIST(Point DFLOAT) := empty()
```

```
  while ((x := readIfCan!(f)) case DFLOAT) repeat
    y : DFLOAT := read!(f)
    lp := cons(point [x,y]$(Point DFLOAT), lp)
    lp
  close!(f)
  drawPoints(lp)
  drawLines(lp)
```

This command will actually create the viewport and the graph if the point data
is in the file "*file.data*".

```
plotData2D("file.data", "2D Data Plot")
```

7.1.10 Addendum: Appending a Graph to a Viewport Window Containing a Graph

This section demonstrates how to append a two-dimensional graph to a viewport
already containing other graphs. The default **draw** command places a graph
into the first `GraphImage` slot position of the `TwoDimensionalViewport`.

This graph is in the first slot in its viewport.

```
v1 := draw(sin(x),x=0..2*%pi)
```

So is this graph.

```
v2 := draw(cos(x),x=0..2*%pi, curveColor==light red())
```

The operation **getGraph** retrieves the `GraphImage` $g1$ from the first slot posi-
tion in the viewport $v1$.

```
g1 := getGraph(v1,1)
```

Now **putGraph** places $g1$ into the the second slot position of $v2$.

```
putGraph(v2,g1,2)
```

Display the new `TwoDimensionalViewport` containing both graphs.

```
makeViewport2D(v2)
```

7.2   Three-Dimensional Graphics

The Axiom three-dimensional graphics package provides the ability to

- generate surfaces defined by a function of two real variables

- generate space curves and tubes defined by parametric equations

- generate surfaces defined by parametric equations

These graphs can be modified by using various options, such as calculating points in the spherical coordinate system or changing the polygon grid size of a surface.

7.2.1   Plotting Three-Dimensional Functions of Two Variables

The simplest three-dimensional graph is that of a surface defined by a function of two variables, $z = f(x, y)$.

The general format for drawing a surface defined by a formula $f(x, y)$ of two variables x and y is:

$$\text{draw(f(x,y), x = a..b, y = c..d, } options)$$

where $a..b$ and $c..d$ define the range of x and y, and where *options* prescribes zero or more options as described in 7.2.4 on page 322. An example of an option is $title == "Title of Graph"$. An alternative format involving a function f is also available.

The simplest way to plot a function of two variables is to use a formula. With formulas you always precede the range specifications with the variable name and an = sign.

```
draw(cos(x*y),x=-3..3,y=-3..3)
```

If you intend to use a function more than once, or it is long and complex, then first give its definition to Axiom.

```
f(x,y) == sin(x)*cos(y)
```

                                                                    Type: Void

To draw the function, just give its name and drop the variables from the range specifications. Axiom compiles your function for efficient computation of data for the graph. Notice that Axiom uses the text of your function as a default title.

```
draw(f,-%pi..%pi,-%pi..%pi)
```

7.2.2   Plotting Three-Dimensional Parametric Space Curves

A second kind of three-dimensional graph is a three-dimensional space curve defined by the parametric equations for $x(t)$, $y(t)$, and $z(t)$ as a function of an independent variable t.

The general format for drawing a three-dimensional space curve defined by parametric formulas $x = f(t)$, $y = g(t)$, and $z = h(t)$ is:

$$\texttt{draw(curve(f(t),g(t),h(t)), t = a..b, } \textit{options)}$$

where $a..b$ defines the range of the independent variable t, and where *options* prescribes zero or more options as described in 7.2.4 on page 322. An example of an option is $title == "Title of Graph"$. An alternative format involving functions f, g and h is also available.

If you use explicit formulas to draw a space curve, always precede the range specification with the variable name and an = sign.

```
draw(curve(5*cos(t), 5*sin(t),t), t=-12..12)
```

Alternatively, you can draw space curves by referring to functions.

```
i1(t:DFLOAT):DFLOAT == sin(t)*cos(3*t/5)
```

```
    Function declaration i1 : DoubleFloat -> DoubleFloat has been added
        to workspace.
```

                                                              Type: Void

This is useful if the functions are to be used more than once . . .

```
i2(t:DFLOAT):DFLOAT == cos(t)*cos(3*t/5)
```

```
Function declaration i2 : DoubleFloat -> DoubleFloat has been added
    to workspace.
```

                                                        Type: Void

or if the functions are long and complex.

```
i3(t:DFLOAT):DFLOAT == cos(t)*sin(3*t/5)
```

```
Function declaration i3 : DoubleFloat -> DoubleFloat has been added
    to workspace.
```

                                                        Type: Void

Give the names of the functions and drop the variable name specification in the
second argument. Again, Axiom supplies a default title.

```
draw(curve(i1,i2,i3),0..15*%pi)
```

7.2.3   Plotting Three-Dimensional Parametric Surfaces

A third kind of three-dimensional graph is a surface defined by parametric equa-
tions for $x(u, v)$, $y(u, v)$, and $z(u, v)$ of two independent variables u and v.

The general format for drawing a three-dimensional graph defined by para-
metric formulas $x = f(u, v)$, $y = g(u, v)$, and $z = h(u, v)$ is:

```
draw(surface(f(u,v),g(u,v),h(u,v)), u = a..b, v = c..d, options)
```

where $a..b$ and $c..d$ define the range of the independent variables u and v,
and where *options* prescribes zero or more options as described in 7.2.4
on page 322. An example of an option is $title == "Title of Graph"$. An
alternative format involving functions f, g and h is also available.

This example draws a graph of a surface plotted using the parabolic cylindrical
coordinate system option. The values of the functions supplied to **surface** are
interpreted in coordinates as given by a `coordinates` option, here as parabolic
cylindrical coordinates (see 7.2.7 on page 331).

```
draw(surface(u*cos(v), u*sin(v), v*cos(u)), u=-4..4, v=0..%pi,
coordinates== parabolicCylindrical)
```

Again, you can graph these parametric surfaces using functions, if the functions are long and complex.

Here we declare the types of arguments and values to be of type `DoubleFloat`.

```
n1(u:DFLOAT,v:DFLOAT):DFLOAT == u*cos(v)
```

    Function declaration n1 : DoubleFloat -> DoubleFloat has been added
        to workspace.

                                                         Type: Void

As shown by previous examples, these declarations are necessary.

```
n2(u:DFLOAT,v:DFLOAT):DFLOAT == u*sin(v)
```

    Function declaration n2 : DoubleFloat -> DoubleFloat has been added
        to workspace.

                                                         Type: Void

In either case, Axiom compiles the functions when needed to graph a result.

```
n3(u:DFLOAT,v:DFLOAT):DFLOAT == u
```

    Function declaration n3 : DoubleFloat -> DoubleFloat has been added
        to workspace.

                                                         Type: Void

Without these declarations, you have to suffix floats with $@DFLOAT$ to get a `DoubleFloat` result. However, a call here with an unadorned float produces a `DoubleFloat`.

```
n3(0.5,1.0)
```

    Compiling function n3 with type (DoubleFloat,DoubleFloat) ->
        DoubleFloat

                                                     Type: DoubleFloat

Draw the surface by referencing the function names, this time choosing the toroidal coordinate system.

```
draw(surface(n1,n2,n3), 1..4, 1..2*%pi, coordinates ==
toroidal(1$DFLOAT))
```

7.2.4   Three-Dimensional Options

The **draw** commands optionally take an optional list of options such as `coordinates` as shown in the last example. Each option is given by the syntax: *name* `==` *value*. Here is a list of the available options in the order that they are described below:

title	coordinates	var1Steps
style	tubeRadius	var2Steps
colorFunction	tubePoints	space

The option *title* gives your graph a title.

```
draw(cos(x*y),x=0..2*%pi,y=0..%pi,title == "Title of Graph")
```

The *style* determines which of four rendering algorithms is used for the graph. The choices are `"wireMesh"`, `"solid"`, `"shade"`, and `"smooth"`.

```
draw(cos(x*y),x=-3..3,y=-3..3, style=="smooth", title=="Smooth
Option")
```

In all but the wire-mesh style, polygons in a surface or tube plot are normally colored in a graph according to their z-coordinate value. Space curves are colored according to their parametric variable value. To change this, you can give a coloring function. The coloring function is sampled across the range of its arguments, then normalized onto the standard Axiom colormap.

A function of one variable makes the color depend on the value of the parametric variable specified for a tube plot.

```
color1(t) == t
```

                                                              Type: Void

```
draw(curve(sin(t), cos(t),0), t=0..2*%pi, tubeRadius == .3,
colorFunction == color1)
```

A function of two variables makes the color depend on the values of the independent variables.

```
color2(u,v) == u**2 - v**2
```

                                                              Type: Void

Use the option `colorFunction` for special coloring.

```
draw(cos(u*v), u=-3..3, v=-3..3, colorFunction == color2)
```

With a three variable function, the color also depends on the value of the function.

```
color3(x,y,fxy) == sin(x*fxy) + cos(y*fxy)
```

```
                                                    Type:  Void
```

```
draw(cos(x*y), x=-3..3, y=-3..3, colorFunction == color3)
```

Normally the Cartesian coordinate system is used. To change this, use the `coordinates` option. For details, see .

```
m(u:DFLOAT,v:DFLOAT):DFLOAT == 1
```

```
   Function declaration m : (DoubleFloat,DoubleFloat) -> DoubleFloat
      has been added to workspace.
```

```
                                                    Type:  Void
```

Use the spherical coordinate system.

```
draw(m, 0..2*%pi,0..%pi, coordinates == spherical,
style=="shade")
```

Space curves may be displayed as tubes with polygonal cross sections. Two options, `tubeRadius` and `tubePoints`, control the size and shape of this cross section.

The `tubeRadius` option specifies the radius of the tube that encircles the specified space curve.

```
draw(curve(sin(t),cos(t),0),t=0..2*%pi, style=="shade",
tubeRadius == .3)
```

The `tubePoints` option specifies the number of vertices defining the polygon that is used to create a tube around the specified space curve. The larger this number is, the more cylindrical the tube becomes.

```
draw(curve(sin(t), cos(t), 0), t=0..2*%pi, style=="shade",
tubeRadius == .25, tubePoints == 3)
```

Options **var1Steps** and **var2Steps** specify the number of intervals into which
the grid defining a surface plot is subdivided with respect to the first and second
parameters of the surface function(s).

```
draw(cos(x*y),x=-3..3,y=-3..3, style=="shade", var1Steps == 30,
var2Steps == 30)
```

The `space` option of a **draw** command lets you build multiple graphs in three
space. To use this option, first create an empty three-space object, then use the
`space` option thereafter. There is no restriction as to the number or kinds of
graphs that can be combined this way.

Create an empty three-space object.

```
s := create3Space()$(ThreeSpace DFLOAT)
```

$$3 - Spacewith0\mathrm{components}$$

                                                 Type: ThreeSpace DoubleFloat

```
m(u:DFLOAT,v:DFLOAT):DFLOAT == 1
```

   Function declaration m : (DoubleFloat,DoubleFloat) -> DoubleFloat
       has been added to workspace.

                                                            Type: Void

Add a graph to this three-space object. The new graph destructively inserts the
graph into s.

```
draw(m,0..%pi,0..2*%pi, coordinates == spherical, space == s)
```

Add a second graph to s.

```
v := draw(curve(1.5*sin(t), 1.5*cos(t),0), t=0..2*%pi, tubeRadius
== .25, space == s)
```

A three-space object can also be obtained from an existing three-dimensional
viewport using the **subspace** command. You can then use **makeViewport3D**
to create a viewport window.

Assign to *subsp* the three-space object in viewport v.

```
subsp := subspace v
```

Reset the space component of *v* to the value of *subsp*.

```
subspace(v, subsp)
```

Create a viewport window from a three-space object.

```
makeViewport3D(subsp,"Graphs")
```

7.2.5 The makeObject Command

An alternate way to create multiple graphs is to use **makeObject**. The **makeObject** command is similar to the **draw** command, except that it returns a three-space object rather than a `ThreeDimensionalViewport`. In fact, **makeObject** is called by the **draw** command to create the `ThreeSpace` then **makeViewport3D** to create a viewport window.

```
m(u:DFLOAT,v:DFLOAT):DFLOAT == 1
```

```
   Function declaration m : (DoubleFloat,DoubleFloat) -> DoubleFloat
      has been added to workspace.
```

$$\text{Type:} \quad \text{Void}$$

Do the last example a new way. First use **makeObject** to create a three-space object *sph*.

```
sph := makeObject(m, 0..%pi, 0..2*%pi, coordinates==spherical)
```

```
   Compiling function m with type (DoubleFloat,DoubleFloat) ->
      DoubleFloat
```

$$3 - Space with 1 component$$

$$\text{Type:} \quad \text{ThreeSpace DoubleFloat}$$

Add a second object to *sph*.

```
makeObject(curve(1.5*sin(t), 1.5*cos(t), 0), t=0..2*%pi, space ==
sph, tubeRadius == .25)
```

```
Compiling function %D with type DoubleFloat -> DoubleFloat
Compiling function %F with type DoubleFloat -> DoubleFloat
Compiling function %H with type DoubleFloat -> DoubleFloat
```

$$3 - Space with 2 components$$

Type: ThreeSpace DoubleFloat

Create and display a viewport containing *sph*.

```
makeViewport3D(sph,"Multiple Objects")
```

Note that an undefined `ThreeSpace` parameter declared in a **makeObject** or **draw** command results in an error. Use the **create3Space** function to define a `ThreeSpace`, or obtain a `ThreeSpace` that has been previously generated before including it in a command line.

7.2.6   Building Three-Dimensional Objects From Primitives

Rather than using the **draw** and **makeObject** commands, you can create three-dimensional graphs from primitives. Operation **create3Space** creates a three-space object to which points, curves and polygons can be added using the operations from the `ThreeSpace` domain. The resulting object can then be displayed in a viewport using **makeViewport3D**.

Create the empty three-space object *space*.

```
space := create3Space()$(ThreeSpace DFLOAT)
```

$$3 - Space with 0 components$$

Type: ThreeSpace DoubleFloat

Objects can be sent to this *space* using the operations exported by the `ThreeSpace` domain. The following examples place curves into *space*.

Add these eight curves to the space.

```
closedCurve(space,[ [0,30,20], [0,30,30], [0,40,30], [0,40,100],
[0,30,100],[0,30,110], [0,60,110], [0,60,100], [0,50,100],
[0,50,30], [0,60,30], [0,60,20] ])
```

$$3 - Space with 1 component$$

                                        Type: ThreeSpace DoubleFloat

```
closedCurve(space,[ [80,0,30], [80,0,100], [70,0,110],
[40,0,110], [30,0,100], [30,0,90], [40,0,90], [40,0,95],
[45,0,100], [65,0,100], [70,0,95], [70,0,35] ])
```

$$3 - Space with 2 components$$

                                        Type: ThreeSpace DoubleFloat

```
closedCurve(space,[ [70,0,35], [65,0,30], [45,0,30], [40,0,35],
[40,0,60], [50,0,60], [50,0,70], [30,0,70], [30,0,30], [40,0,20],
[70,0,20], [80,0,30] ])
```

$$3 - Space with 3 components$$

                                        Type: ThreeSpace DoubleFloat

```
closedCurve(space,[ [0,70,20], [0,70,110], [0,110,110],
[0,120,100], [0,120,70], [0,115,65], [0,120,60], [0,120,30],
[0,110,20], [0,80,20], [0,80,30], [0,80,20] ])
```

$$3 - Space with 4 components$$

                                        Type: ThreeSpace DoubleFloat

```
closedCurve(space,[ [0,105,30], [0,110,35], [0,110,55],
[0,105,60], [0,80,60], [0,80,70], [0,105,70], [0,110,75],
[0,110,95], [0,105,100], [0,80,100], [0,80,20], [0,80,30] ])
```

$$3 - Space with 5 components$$

                                        Type: ThreeSpace DoubleFloat

```
closedCurve(space,[ [140,0,20], [140,0,110], [130,0,110],
[90,0,20], [101,0,20],[114,0,50], [130,0,50], [130,0,60],
[119,0,60], [130,0,85], [130,0,20] ])
```

$$3 - Space with 6 components$$

<div align="right">Type: ThreeSpace DoubleFloat</div>

```
closedCurve(space,[ [0,140,20], [0,140,110], [0,150,110],
[0,170,50], [0,190,110], [0,200,110], [0,200,20], [0,190,20],
[0,190,75], [0,175,35], [0,165,35],[0,150,75], [0,150,20] ])
```

$$3 - Space with 7 components$$

<div align="right">Type: ThreeSpace DoubleFloat</div>

```
closedCurve(space,[ [200,0,20], [200,0,110], [189,0,110],
[160,0,45], [160,0,110], [150,0,110], [150,0,20], [161,0,20],
[190,0,85], [190,0,20] ])
```

$$3 - Space with 8 components$$

<div align="right">Type: ThreeSpace DoubleFloat</div>

Create and display the viewport using **makeViewport3D**. Options may also be given but here are displayed as a list with values enclosed in parentheses.

```
makeViewport3D(space, title == "Letters")
```

Cube Example

As a second example of the use of primitives, we generate a cube using a polygon mesh. It is important to use a consistent orientation of the polygons for correct generation of three-dimensional objects.

Again start with an empty three-space object.

```
spaceC := create3Space()$(ThreeSpace DFLOAT)
```

$$3 - Space with 0 components$$

<div align="right">Type: ThreeSpace DoubleFloat</div>

For convenience, give `DoubleFloat` values $+1$ and -1 names.

```
x:  DFLOAT := 1
```

<div align="center">1.0</div>

<div align="right">Type: DoubleFloat</div>

```
y:  DFLOAT := -1
```

$$-1.0$$

<div align="right">Type: DoubleFloat</div>

Define the vertices of the cube.

```
a := point [x,x,y,1::DFLOAT]$(Point DFLOAT)
```

$$[1.0, 1.0, -1.0, 1.0]$$

<div align="right">Type: Point DoubleFloat</div>

```
b := point [y,x,y,4::DFLOAT]$(Point DFLOAT)
```

$$[-1.0, 1.0, -1.0, 4.0]$$

<div align="right">Type: Point DoubleFloat</div>

```
c := point [y,x,x,8::DFLOAT]$(Point DFLOAT)
```

$$[-1.0, 1.0, 1.0, 8.0]$$

<div align="right">Type: Point DoubleFloat</div>

```
d := point [x,x,x,12::DFLOAT]$(Point DFLOAT)
```

$$[1.0, 1.0, 1.0, 12.0]$$

<div align="right">Type: Point DoubleFloat</div>

```
e := point [x,y,y,16::DFLOAT]$(Point DFLOAT)
```

$$[1.0, -1.0, -1.0, 16.0]$$

<div align="right">Type: Point DoubleFloat</div>

```
f := point [y,y,y,20::DFLOAT]$(Point DFLOAT)
```

$$[-1.0, -1.0, -1.0, 20.0]$$

<div align="right">Type: Point DoubleFloat</div>

```
g := point [y,y,x,24::DFLOAT]$(Point DFLOAT)
```

$$[-1.0, -1.0, 1.0, 24.0]$$

<div align="right">Type: Point DoubleFloat</div>

```
h := point [x,y,x,27::DFLOAT]$(Point DFLOAT)
```

$$[1.0, -1.0, 1.0, 27.0]$$

<div align="right">Type: Point DoubleFloat</div>

Add the faces of the cube as polygons to the space using a consistent orientation.

```
polygon(spaceC,[d,c,g,h])
```

$$3 - Space with 1 \text{component}$$

<div align="right">Type: ThreeSpace DoubleFloat</div>

```
polygon(spaceC,[d,h,e,a])
```

$$3 - Space with 2 \text{components}$$

<div align="right">Type: ThreeSpace DoubleFloat</div>

```
polygon(spaceC,[c,d,a,b])
```

$$3 - Space with 3 \text{components}$$

<div align="right">Type: ThreeSpace DoubleFloat</div>

```
polygon(spaceC,[g,c,b,f])
```

$$3 - Spacewith4\text{components}$$

            Type: ThreeSpace DoubleFloat

```
polygon(spaceC,[h,g,f,e])
```

$$3 - Spacewith5\text{components}$$

            Type: ThreeSpace DoubleFloat

```
polygon(spaceC,[e,f,b,a])
```

$$3 - Spacewith6\text{components}$$

            Type: ThreeSpace DoubleFloat

Create and display the viewport.

```
makeViewport3D(spaceC, title == "Cube")
```

7.2.7   Coordinate System Transformations

The `CoordinateSystems` package provides coordinate transformation functions
that map a given data point from the coordinate system specified into the
Cartesian coordinate system. The default coordinate system, given a triplet
$(f(u,v),u,v)$, assumes that $z = f(u,v)$, $x = u$ and $y = v$, that is, reads the
coordinates in (z,x,y) order.

```
m(u:DFLOAT,v:DFLOAT):DFLOAT == u**2
```

   Function declaration m : (DoubleFloat,DoubleFloat) -> DoubleFloat
     has been added to workspace.

            Type: Void

Graph plotted in default coordinate system.

```
draw(m,0..3,0..5)
```

The z coordinate comes first since the first argument of the **draw** command gives its values. In general, the coordinate systems Axiom provides, or any that you make up, must provide a map to an (x, y, z) triplet in order to be compatible with the **coordinates** `DrawOption`. Here is an example.

Define the identity function.

```
cartesian(point:Point DFLOAT):Point DFLOAT == point
```

```
   Function declaration cartesian : Point DoubleFloat -> Point
      DoubleFloat has been added to workspace.
```

                                                           Type: Void

Pass *cartesian* as the **coordinates** parameter to the **draw** command.

```
draw(m,0..3,0..5,coordinates==cartesian)
```

What happened? The option `coordinates == cartesian` directs Axiom to treat the dependent variable m defined by $m = u^2$ as the x coordinate. Thus the triplet of values (m, u, v) is transformed to coordinates (x, y, z) and so we get the graph of $x = y^2$.

Here is another example. The **cylindrical** transform takes input of the form (w, u, v), interprets it in the order (r, θ, z) and maps it to the Cartesian coordinates $x = r \cos(\theta)$, $y = r \sin(\theta)$, $z = z$ in which r is the radius, θ is the angle and z is the z-coordinate.

An example using the **cylindrical** coordinates for the constant $r = 3$.

```
f(u:DFLOAT,v:DFLOAT):DFLOAT == 3
```

```
   Function declaration f : (DoubleFloat,DoubleFloat) -> DoubleFloat
      has been added to workspace.
```

                                                           Type: Void

Graph plotted in cylindrical coordinates.

```
draw(f,0..%pi,0..6,coordinates==cylindrical)
```

Suppose you would like to specify z as a function of r and θ instead of just r? Well, you still can use the **cylindrical** Axiom transformation but we have to reorder the triplet before passing it to the transformation.

First, let's create a point to work with and call it *pt* with some color *col*.

```
col := 5
```

$$5$$

<div align="right">Type: PositiveInteger</div>

```
pt := point[1,2,3,col]$(Point DFLOAT)
```

$$[1.0, 2.0, 3.0, 5.0]$$

<div align="right">Type: Point DoubleFloat</div>

The reordering you want is (z, r, θ) to (r, θ, z) so that the first element is moved to the third element, while the second and third elements move forward and the color element does not change.

Define a function **reorder** to reorder the point elements.

```
reorder(p:Point DFLOAT):Point DFLOAT == point[p.2, p.3, p.1, p.4]
```

    Function declaration reorder : Point DoubleFloat -> Point
        DoubleFloat has been added to workspace.

<div align="right">Type: Void</div>

The function moves the second and third elements forward but the color does not change.

```
reorder pt
```

$$[2.0, 3.0, 1.0, 5.0]$$

<div align="right">Type: Point DoubleFloat</div>

The function **newmap** converts our reordered version of the cylindrical coordinate system to the standard (x, y, z) Cartesian system.

```
newmap(pt:Point DFLOAT):Point DFLOAT == cylindrical(reorder pt)
```

    Function declaration newmap : Point DoubleFloat -> Point DoubleFloat
        has been added to workspace.

```
                                               Type:  Void
```

```
newmap pt
```

$$[-1.9799849932008908, 0.28224001611973443, 1.0, 5.0]$$

```
                                    Type:  Point DoubleFloat
```

Graph the same function f using the coordinate mapping of the function *newmap*, so it is now interpreted as $z = 3$:

```
draw(f,0..3,0..2*%pi,coordinates==newmap)
```

The `CoordinateSystems` package exports the following operations: **bipolar**, **bipolarCylindrical**, **cartesian**, **conical**, **cylindrical**, **elliptic**, **ellipticCylindrical**, **oblateSpheroidal**, **parabolic**, **parabolicCylindrical**, **paraboloidal**, **polar**, **prolateSpheroidal**, **spherical**, and **toroidal**. Use Browse or the)`show` system command to get more information.

7.2.8   Three-Dimensional Clipping

A three-dimensional graph can be explicitly clipped within the **draw** command by indicating a minimum and maximum threshold for the given function definition. These thresholds can be defined using the Axiom **min** and **max** functions.

```
gamma(x,y) ==
  g := Gamma complex(x,y)
  point [x, y, max( min(real g, 4), -4), argument g]
```

Here is an example that clips the gamma function in order to eliminate the extreme divergence it creates.

```
draw(gamma,-%pi..%pi,-%pi..%pi,var1Steps==50,var2Steps==50)
```

7.2.9   Three-Dimensional Control-Panel

Once you have created a viewport, move your mouse to the viewport and click with your left mouse button. This displays a control-panel on the side of the viewport that is closest to where you clicked.
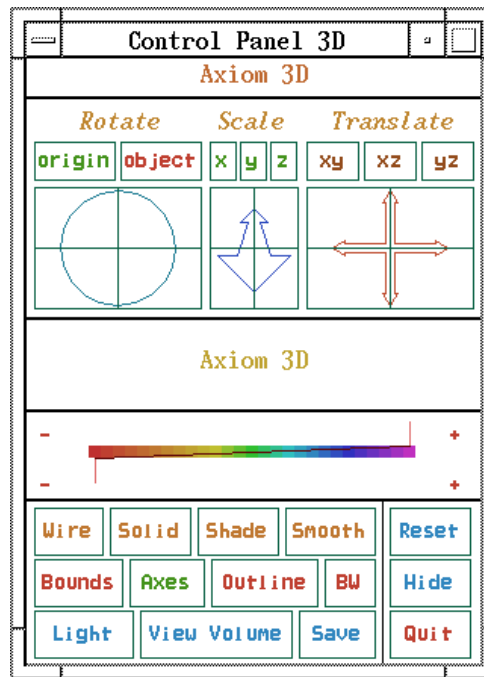
Figure 7.3: Three-dimensional control-panel.

Transformations

We recommend you first select the **Bounds** button while executing transformations since the bounding box displayed indicates the object's position as it changes.

Rotate: A rotation transformation occurs by clicking the mouse within the **Rotate** window in the upper left corner of the control-panel. The rotation is computed in spherical coordinates, using the horizontal mouse position to increment or decrement the value of the longitudinal angle θ within the range of 0 to 2π and the vertical mouse position to increment or decrement the value of the latitudinal angle ϕ within the range of $-\pi$ to π. The active mode of rotation is displayed in green on a color monitor or in clear text on a black and white monitor, while the inactive mode is displayed in red for color display or a mottled pattern for black and white.

> **origin:** The **origin** button indicates that the rotation is to occur with respect to the origin of the viewing space, that is indicated by the axes.
>
> **object:** The **object** button indicates that the rotation is to occur with respect to the center of volume of the object, independent of the axes' origin position.

Scale: A scaling transformation occurs by clicking the mouse within the **Scale** window in the upper center of the control-panel, containing a zoom arrow. The axes along which the scaling is to occur are indicated by selecting the appropriate button above the zoom arrow window. The selected axes are displayed in green on a color monitor or in clear text on a black and white monitor, while the unselected axes are displayed in red for a color display or a mottled pattern for black and white.

> **uniform:** Uniform scaling along the x, y and z axes occurs when all the axes buttons are selected.

> **non-uniform:** If any of the axes buttons are not selected, non-uniform scaling occurs, that is, scaling occurs only in the direction of the axes that are selected.

Translate: Translation occurs by indicating with the mouse in the **Translate** window the direction you want the graph to move. This window is located in the upper right corner of the control-panel and contains a potentiometer with crossed arrows pointing up, down, left and right. Along the top of the **Translate** window are three buttons (**XY**, **XZ**, and **YZ**) indicating the three orthographic projection planes. Each orientates the group as a view into that plane. Any translation of the graph occurs only along this plane.

Messages

The window directly below the potentiometer windows for transformations is used to display system messages relating to the viewport, the control-panel and the current graph displaying status.

Colormap

Directly below the message window is the colormap range indicator window. The Axiom Colormap shows a sampling of the spectrum from which hues can be drawn to represent the colors of a surface. The Colormap is composed of five shades for each of the hues along this spectrum. By moving the markers above and below the Colormap, the range of hues that are used to color the existing surface are set. The bottom marker shows the hue for the low end of the color range and the top marker shows the hue for the upper end of the range. Setting the bottom and top markers at the same hue results in monochromatic smooth shading of the graph when **Smooth** mode is selected. At each end of the Colormap are **+** and **-** buttons. When clicked on, these increment or decrement the top or bottom marker.

Buttons

Below the Colormap window and to the left are located various buttons that determine the characteristics of a graph. The buttons along the bottom and right hand side all have special meanings; the remaining buttons in the first row indicate the mode or style used to display the graph. The second row are toggles that turn on or off a property of the graph. On a color monitor, the property is on if green (clear text, on a monochrome monitor) and off if red (mottled pattern, on a monochrome monitor). Here is a list of their functions.

Wire displays surface and tube plots as a wireframe image in a single color (blue) with no hidden surfaces removed, or displays space curve plots in colors based upon their parametric variables. This is the fastest mode for displaying a graph. This is very useful when you want to find a good orientation of your graph.

Solid displays the graph with hidden surfaces removed, drawing each polygon beginning with the furthest from the viewer. The edges of the polygons are displayed in the hues specified by the range in the Colormap window.

Shade displays the graph with hidden surfaces removed and with the polygons shaded, drawing each polygon beginning with the furthest from the viewer. Polygons are shaded in the hues specified by the range in the Colormap window using the Phong illumination model.

Smooth displays the graph using a renderer that computes the graph one line at a time. The location and color of the graph at each visible point on the screen are determined and displayed using the Phong illumination model. Smooth shading is done in one of two ways, depending on the range selected in the colormap window and the number of colors available from the hardware and/or window manager. When the top and bottom markers of the colormap range are set to different hues, the graph is rendered by dithering between the transitions in color hue. When the top and bottom markers of the colormap range are set to the same hue, the graph is rendered using the Phong smooth shading model. However, if enough colors cannot be allocated for this purpose, the renderer reverts to the color dithering method until a sufficient color supply is available. For this reason, it may not be possible to render multiple Phong smooth shaded graphs at the same time on some systems.

Bounds encloses the entire volume of the viewgraph within a bounding box, or removes the box if previously selected. The region that encloses the entire volume of the viewport graph is displayed.

Axes displays Cartesian coordinate axes of the space, or turns them off if previously selected.

Outline causes quadrilateral polygons forming the graph surface to be outlined in black when the graph is displayed in **Shade** mode.

BW converts a color viewport to black and white, or vice-versa. When this
button is selected the control-panel and viewport switch to an immutable
colormap composed of a range of grey scale patterns or tiles that are used
wherever shading is necessary.

Light takes you to a control-panel described below.

ViewVolume takes you to another control-panel as described below.

Save creates a menu of the possible file types that can be written using the
control-panel. The **Exit** button leaves the save menu. The **Pixmap** but-
ton writes an Axiom pixmap of the current viewport contents. The file
is called **axiom3D.pixmap** and is located in the directory from which
Axiom or **viewAlone** was started. The **PS** button writes the current
viewport contents to PostScript output rather than to the viewport win-
dow. By default the file is called **axiom3D.ps**; however, if a file name is
specified in the user's **.Xdefaults** file it is used. The file is placed in the
directory from which the Axiom or **viewAlone** session was begun. See
also the **write** function.

Reset returns the object transformation characteristics back to their initial
states.

Hide causes the control-panel for the corresponding viewport to disappear from
the screen.

Quit queries whether the current viewport session should be terminated.

Light

The **Light** button changes the control-panel into the **Lighting Control-Panel**.
At the top of this panel, the three axes are shown with the same orientation as
the object. A light vector from the origin of the axes shows the current position
of the light source relative to the object. At the bottom of the panel is an **Abort**
button that cancels any changes to the lighting that were made, and a **Return**
button that carries out the current set of lighting changes on the graph.

XY: The **XY** lighting axes window is below the **Lighting Control-Panel** title
and to the left. This changes the light vector within the **XY** view plane.

Z: The **Z** lighting axis window is below the **Lighting Control-Panel** title and
in the center. This changes the **Z** location of the light vector.

Intensity: Below the **Lighting Control-Panel** title and to the right is the
light intensity meter. Moving the intensity indicator down decreases the
amount of light emitted from the light source. When the indicator is at
the top of the meter the light source is emitting at 100% intensity. At the
bottom of the meter the light source is emitting at a level slightly above
ambient lighting.

View Volume

The **View Volume** button changes the control-panel into the **Viewing Volume Panel**. At the bottom of the viewing panel is an **Abort** button that cancels any changes to the viewing volume that were made and a *Return* button that carries out the current set of viewing changes to the graph.

Eye Reference: At the top of this panel is the **Eye Reference** window. It shows a planar projection of the viewing pyramid from the eye of the viewer relative to the location of the object. This has a bounding region represented by the rectangle on the left. Below the object rectangle is the **Hither** window. By moving the slider in this window the hither clipping plane sets the front of the view volume. As a result of this depth clipping all points of the object closer to the eye than this hither plane are not shown. The **Eye Distance** slider to the right of the **Hither** slider is used to change the degree of perspective in the image.

Clip Volume: The **Clip Volume** window is at the bottom of the **Viewing Volume Panel**. On the right is a **Settings** menu. In this menu are buttons to select viewing attributes. Selecting the **Perspective** button computes the image using perspective projection. The **Show Region** button indicates whether the clipping region of the volume is to be drawn in the viewport and the **Clipping On** button shows whether the view volume clipping is to be in effect when the image is drawn. The left side of the **Clip Volume** window shows the clipping boundary of the graph. Moving the knobs along the **X**, **Y**, and **Z** sliders adjusts the volume of the clipping region accordingly.

7.2.10   Operations for Three-Dimensional Graphics

Here is a summary of useful Axiom operations for three-dimensional graphics. Each operation name is followed by a list of arguments. Each argument is written as a variable informally named according to the type of the argument (for example, *integer*). If appropriate, a default value for an argument is given in parentheses immediately following the name.

adaptive3D? ()
> tests whether space curves are to be plotted according to the adaptive refinement algorithm.

axes (*viewport, string*(`"on"`))
> turns the axes on and off.

close (*viewport*)
> closes the viewport.

colorDef (*viewport, color$_1$*(`1`)*, color$_2$*(`27`))
    sets the colormap range to be from *color$_1$* to *color$_2$*.

controlPanel (*viewport, string*(`"off"`))
    declares whether the control-panel for the viewport is to be displayed or
    not.

diagonals (*viewport, string*(`"off"`))
    declares whether the polygon outline includes the diagonals or not.

drawStyle (*viewport, style*)
    selects which of four drawing styles are used: `"wireMesh"`, `"solid"`,
    `"shade"`, or `"smooth"`.

eyeDistance (*viewport,float*(`500`))
    sets the distance of the eye from the origin of the object for use in the
    perspective.

key (*viewport*)
    returns the operating system process ID number for the viewport.

lighting (*viewport, float$_x$*(`-0.5`)*, float$_y$*(`0.5`)*, float$_z$*(`0.5`))
    sets the Cartesian coordinates of the light source.

modifyPointData (*viewport,integer,point*)
    replaces the coordinates of the point with the index *integer* with *point*.

move (*viewport, integer$_x$*(`viewPosDefault`)*, integer$_y$*(`viewPosDefault`))
    moves the upper left-hand corner of the viewport to screen position
    (*integer$_x$, integer$_y$*).

options (*viewport*)
    returns a list of all current draw options.

outlineRender (*viewport, string*(`"off"`))
    turns polygon outlining off or on when drawing in `"shade"` mode.

perspective (*viewport, string*(`"on"`))
    turns perspective viewing on and off.

reset (*viewport*)
    resets the attributes of a viewport to their initial settings.

resize (*viewport,   integer$_{width}$ (`viewSizeDefault`),   integer$_{height}$*
    (`viewSizeDefault`))
    resets the width and height values for a viewport.

rotate (*viewport, number$_\theta$*(`viewThetaDefapult`)*, number$_\phi$*(`viewPhiDefault`))
    rotates the viewport by rotation angles for longitude (θ) and latitude (ϕ).
    Angles designate radians if given as floats, or degrees if given as integers.

setAdaptive3D (*boolean*(`true`))
>   sets whether space curves are to be plotted according to the adaptive refinement algorithm.

setMaxPoints3D (*integer*(`1000`))
>   sets the default maximum number of possible points to be used when constructing a three-dimensional space curve.

setMinPoints3D (*integer*(`49`))
>   sets the default minimum number of possible points to be used when constructing a three-dimensional space curve.

setScreenResolution3D (*integer*(`49`))
>   sets the default screen resolution constant used in setting the computation limit of adaptively generated three-dimensional space curve plots.

showRegion (*viewport, string*(`"off"`))
>   declares whether the bounding box of a graph is shown or not.

subspace (*viewport*)
>   returns the space component.

subspace (*viewport, subspace*)
>   resets the space component to *subspace.*

title (*viewport, string*)
>   gives the viewport the title *string.*

translate (*viewport, $float_x$*(`viewDeltaXDefault`)*, $float_y$*(`viewDeltaYDefault`))
>   translates the object horizontally and vertically relative to the center of the viewport.

intensity (*viewport,float*(`1.0`))
>   resets the intensity I of the light source, $0 \leq I \leq 1$.

tubePointsDefault ([*integer*(`6`)])
>   sets or indicates the default number of vertices defining the polygon that is used to create a tube around a space curve.

tubeRadiusDefault ([*float*(`0.5`)])
>   sets or indicates the default radius of the tube that encircles a space curve.

var1StepsDefault ([*integer*(`27`)])
>   sets or indicates the default number of increments into which the grid defining a surface plot is subdivided with respect to the first parameter declared in the surface function.

var2StepsDefault ([*integer*(`27`)])
>   sets or indicates the default number of increments into which the grid defining a surface plot is subdivided with respect to the second parameter declared in the surface function.

viewDefaults ([$integer_{point}$, $integer_{line}$, $integer_{axes}$, $integer_{units}$, $float_{point}$, $list_{position}$, $list_{size}$])

> resets the default settings for the point color, line color, axes color, units color, point size, viewport upper left-hand corner position, and the viewport size.

viewDeltaXDefault ([$float$(0)])

> resets the default horizontal offset from the center of the viewport, or returns the current default offset if no argument is given.

viewDeltaYDefault ([$float$(0)])

> resets the default vertical offset from the center of the viewport, or returns the current default offset if no argument is given.

viewPhiDefault ([$float$(-π/4)])

> resets the default latitudinal view angle, or returns the current default angle if no argument is given. ϕ is set to this value.

viewpoint ($viewport$, $float_x$, $float_y$, $float_z$)

> sets the viewing position in Cartesian coordinates.

viewpoint ($viewport$, $float_\theta$, $Float_\phi$)

> sets the viewing position in spherical coordinates.

viewpoint ($viewport$, $Float_\theta$, $Float_\phi$, $Float_{scaleFactor}$, $Float_{xOffset}$, $Float_{yOffset}$)

> sets the viewing position in spherical coordinates, the scale factor, and offsets. θ (longitude) and ϕ (latitude) are in radians.

viewPosDefault ([$list$([0,0])])

> sets or indicates the position of the upper left-hand corner of a two-dimensional viewport, relative to the display root window (the upper left-hand corner of the display is $[0, 0]$).

viewSizeDefault ([$list$([400,400])])

> sets or indicates the width and height dimensions of a viewport.

viewThetaDefault ([$float$(π/4)])

> resets the default longitudinal view angle, or returns the current default angle if no argument is given. When a parameter is specified, the default longitudinal view angle θ is set to this value.

viewWriteAvailable ([$list$(["pixmap", "bitmap", "postscript", "image"])])

> indicates the possible file types that can be created with the **write** function.

viewWriteDefault ([$list$([])])

> sets or indicates the default types of files that are created in addition to the **data** file when a **write** command is executed on a viewport.

viewScaleDefault ([*float*])
> sets the default scaling factor, or returns the current factor if no argument is given.

write (*viewport, directory,* [*option*])
> writes the file **data** for *viewport* in the directory *directory*. An optional third argument specifies a file type (one of `pixmap`, `bitmap`, `postscript`, or `image`), or a list of file types. An additional file is written for each file type listed.

scale (*viewport, float*(`2.5`))
> specifies the scaling factor.

7.2.11 Customization using .Xdefaults

Both the two-dimensional and three-dimensional drawing facilities consult the **.Xdefaults** file for various defaults. The list of defaults that are recognized by the graphing routines is discussed in this section. These defaults are preceded by `Axiom.3D.` for three-dimensional viewport defaults, `Axiom.2D.` for two-dimensional viewport defaults, or `Axiom*` (no dot) for those defaults that are acceptable to either viewport type.

`Axiom*buttonFont:` *font*
> This indicates which font type is used for the button text on the control-panel. **Rom11**

`Axiom.2D.graphFont:` *font*   (2D only)
> This indicates which font type is used for displaying the graph numbers and slots in the **Graphs** section of the two-dimensional control-panel. **Rom22**

`Axiom.3D.headerFont:` *font*
> This indicates which font type is used for the axes labels and potentiometer header names on three-dimensional viewport windows. This is also used for two-dimensional control-panels for indicating which font type is used for potentionmeter header names and multiple graph title headers. **Itl14**

`Axiom*inverse:` *switch*
> This indicates whether the background color is to be inverted from white to black. If `on`, the graph viewports use black as the background color. If `off` or no declaration is made, the graph viewports use a white background. **off**

`Axiom.3D.lightingFont:` *font*   (3D only)
> This indicates which font type is used for the **x**, **y**, and **z** labels of the two lighting axes potentiometers, and for the **Intensity** title on the lighting control-panel. **Rom10**

`Axiom.2D.messageFont, Axiom.3D.messageFont:` *font*
>   These indicate the font type to be used for the text in the control-panel message window. **Rom14**

`Axiom*monochrome:` *switch*
>   This indicates whether the graph viewports are to be displayed as if the monitor is black and white, that is, a 1 bit plane. If `on` is specified, the viewport display is black and white. If `off` is specified, or no declaration for this default is given, the viewports are displayed in the normal fashion for the monitor in use. **off**

`Axiom.2D.postScript:` *filename*
>   This specifies the name of the file that is generated when a 2D PostScript graph is saved. **axiom2D.ps**

`Axiom.3D.postScript:` *filename*
>   This specifies the name of the file that is generated when a 3D PostScript graph is saved. **axiom3D.ps**

`Axiom*titleFont` *font*
>   This indicates which font type is used for the title text and, for three-dimensional graphs, in the lighting and viewing-volume control-panel windows. **Rom14**

`Axiom.2D.unitFont:` *font*   (2D only)
>   This indicates which font type is used for displaying the unit labels on two-dimensional viewport graphs. **6x10**

`Axiom.3D.volumeFont:` *font*   (3D only)
>   This indicates which font type is used for the **x**, **y**, and **z** labels of the clipping region sliders; for the **Perspective**, **Show Region**, and **Clipping On** buttons under **Settings**, and above the windows for the **Hither** and **Eye Distance** sliders in the **Viewing Volume Panel** of the three-dimensional control-panel. **Rom8**

Chapter 8

Advanced Problem Solving

In this chapter we describe techniques useful in solving advanced problems with Axiom.

8.1 Numeric Functions

Axiom provides two basic floating-point types: `Float` and `DoubleFloat`. This section describes how to use numerical operations defined on these types and the related complex types. As we mentioned in Chapter 1 on page 71, the `Float` type is a software implementation of floating-point numbers in which the exponent and the significand may have any number of digits. See 9.27 on page 565 for detailed information about this domain. The `DoubleFloat` (see 9.17 on page 533) is usually a hardware implementation of floating point numbers, corresponding to machine double precision. The types `Complex Float` and `Complex DoubleFloat` are the corresponding software implementations of complex floating-point numbers. In this section the term *floating-point type* means any of these four types. The floating-point types implement the basic elementary functions. These include (where `$` means `DoubleFloat`, `Float`, `Complex DoubleFloat`, or `Complex Float`):

exp, **log**: $\$->\$$
sin, **cos**, **tan**, **cot**, **sec**, **csc**: $\$->\$$
sin, **cos**, **tan**, **cot**, **sec**, **csc**: $\$->\$$
asin, **acos**, **atan**, **acot**, **asec**, **acsc**: $\$->\$$
sinh, **cosh**, **tanh**, **coth**, **sech**, **csch**: $\$->\$$
asinh, **acosh**, **atanh**, **acoth**, **asech**, **acsch**: $\$->\$$
pi: $()->\$$
sqrt: $\$->\$$
nthRoot: $(\$, Integer)->\$$
**: $(\$, FractionInteger)->\$$

345

**: $(\$, \$)-> \$$

The handling of roots depends on whether the floating-point type is real or complex: for the real floating-point types, `DoubleFloat` and `Float`, if a real root exists the one with the same sign as the radicand is returned; for the complex floating-point types, the principal value is returned. Also, for real floating-point types the inverse functions produce errors if the results are not real. This includes cases such as $asin(1.2)$, $log(-3.2)$, $sqrt(-1.1)$.

The default floating-point type is `Float` so to evaluate functions using `Float` or `Complex Float`, just use normal decimal notation.

```
exp(3.1)
```

$$22.197951281441633405$$

<div align="right">Type: Float</div>

```
exp(3.1 + 4.5 * %i)
```

$$-4.6792348860969899118 - 21.699165928071731864\ i$$

<div align="right">Type: Complex Float</div>

To evaluate functions using `DoubleFloat` or `Complex DoubleFloat`, a declaration or conversion is required.

```
r:  DFLOAT := 3.1; t:  DFLOAT := 4.5; exp(r + t*%i)
```

$$-4.6792348860969906 - 21.699165928071732\ i$$

<div align="right">Type: Complex DoubleFloat</div>

```
exp(3.1::DFLOAT + 4.5::DFLOAT * %i)
```

$$-4.6792348860969906 - 21.699165928071732\ i$$

<div align="right">Type: Complex DoubleFloat</div>

A number of special functions are provided by the package `DoubleFloatSpecialFunctions` for the machine-precision floating-point types. The special functions provided are listed below, where F stands for

the types `DoubleFloat` and `Complex DoubleFloat`. The real versions of the functions yield an error if the result is not real.

Gamma: $F- > F$
$Gamma(z)$ is the Euler gamma function, $\Gamma(z)$, defined by

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt.$$

Beta: $F- > F$
$Beta(u, v)$ is the Euler Beta function, $B(u, v)$, defined by

$$B(u, v) = \int_0^1 t^{u-1} (1 - t)^{v-1} dt.$$

This is related to $\Gamma(z)$ by

$$B(u, v) = \frac{\Gamma(u)\Gamma(v)}{\Gamma(u + v)}.$$

logGamma: $F- > F$
$logGamma(z)$ is the natural logarithm of $\Gamma(z)$. This can often be computed even if $\Gamma(z)$ cannot.

digamma: $F- > F$
$digamma(z)$, also called $psi(z)$, is the function $\psi(z)$, defined by

$$\psi(z) = \Gamma'(z)/\Gamma(z).$$

polygamma: $(NonNegativeInteger, F)- > F$
$polygamma(n, z)$ is the n-th derivative of $\psi(z)$, written $\psi^{(n)}(z)$.

besselJ: $(F, F)- > F$
$besselJ(v, z)$ is the Bessel function of the first kind, $J_\nu(z)$. This function satisfies the differential equation

$$z^2 w''(z) + z w'(z) + (z^2 - \nu^2) w(z) = 0.$$

besselY: $(F, F)- > F$
$besselY(v, z)$ is the Bessel function of the second kind, $Y_\nu(z)$. This function satisfies the same differential equation as **besselJ**. The implementation simply uses the relation
$$Y_\nu(z) = \frac{J_\nu(z)\cos(\nu\pi) - J_{-\nu}(z)}{\sin(\nu\pi)}.$$

besselI: $(F, F)- > F$
$besselI(v, z)$ is the modified Bessel function of the first kind, $I_\nu(z)$. This function satisfies the differential equation

$$z^2 w''(z) + z w'(z) - (z^2 + \nu^2) w(z) = 0.$$

besselK: $(F, F)- > F$
$besselK(v, z)$ is the modified Bessel function of the second kind, $K_\nu(z)$. This function satisfies the same differential equation as **besselI**. The implementation simply uses the relation

$$K_\nu(z) = \pi \frac{I_{-\nu}(z) - I_\nu(z)}{2\sin(\nu\pi)}.$$

airyAi: $F- > F$
$airyAi(z)$ is the Airy function $Ai(z)$. This function satisfies the differential equation $w''(z) - zw(z) = 0$. The implementation simply uses the relation

$$Ai(-z) = \frac{1}{3}\sqrt{z}(J_{-1/3}(\frac{2}{3}z^{3/2}) + J_{1/3}(\frac{2}{3}z^{3/2})).$$

airyBi: $F- > F$
$airyBi(z)$ is the Airy function $Bi(z)$. This function satisfies the same differential equation as **airyAi**. The implementation simply uses the relation

$$Bi(-z) = \frac{1}{3}\sqrt{3z}(J_{-1/3}(\frac{2}{3}z^{3/2}) - J_{1/3}(\frac{2}{3}z^{3/2})).$$

hypergeometric0F1: $(F, F)- > F$
$hypergeometric0F1(c, z)$ is the hypergeometric function ${}_0F_1(; c; z)$.

The above special functions are defined only for small floating-point types. If you give `Float` arguments, they are converted to `DoubleFloat` by Axiom.

```
Gamma(0.5)**2
```

$$3.14159265358979$$

Type: DoubleFloat

```
a := 2.1; b := 1.1; besselI(a + %i*b, b*a + 1)
```

$$2.489481690673867 - 2.365846713181643 \, i$$

Type: Complex DoubleFloat

A number of additional operations may be used to compute numerical values. These are special polynomial functions that can be evaluated for values in any commutative ring R, and in particular for values in any floating-point type. The following operations are provided by the package `OrthogonalPolynomialFunctions`:

chebyshevT: $(NonNegativeInteger, R) -> R$
chebyshevT(n, z) is the n-th Chebyshev polynomial of the first kind, $T_n(z)$. These are defined by

$$\frac{1 - tz}{1 - 2tz + t^2} = \sum_{n=0}^{\infty} T_n(z)t^n.$$

chebyshevU: $(NonNegativeInteger, R) -> R$
chebyshevU(n, z) is the n-th Chebyshev polynomial of the second kind, $U_n(z)$. These are defined by

$$\frac{1}{1 - 2tz + t^2} = \sum_{n=0}^{\infty} U_n(z)t^n.$$

hermiteH: $(NonNegativeInteger, R) -> R$
hermiteH(n, z) is the n-th Hermite polynomial, $H_n(z)$. These are defined by

$$e^{2tz - t^2} = \sum_{n=0}^{\infty} H_n(z)\frac{t^n}{n!}.$$

laguerreL: $(NonNegativeInteger, R) -> R$
laguerreL(n, z) is the n-th Laguerre polynomial, $L_n(z)$. These are defined by

$$\frac{e^{-\frac{tz}{1-t}}}{1 - t} = \sum_{n=0}^{\infty} L_n(z)\frac{t^n}{n!}.$$

laguerreL: $(NonNegativeInteger, NonNegativeInteger, R) -> R$
laguerreL(m, n, z) is the associated Laguerre polynomial, $L_n^m(z)$. This is the m-th derivative of $L_n(z)$.

legendreP: $(NonNegativeInteger, R) -> R$
legendreP(n, z) is the n-th Legendre polynomial, $P_n(z)$. These are defined by

$$\frac{1}{\sqrt{1 - 2tz + t^2}} = \sum_{n=0}^{\infty} P_n(z)t^n.$$

These operations require non-negative integers for the indices, but otherwise the argument can be given as desired.

```
[chebyshevT(i, z) for i in 0..5]
```

$$\left[1, z, 2\ z^2 - 1, 4\ z^3 - 3\ z, 8\ z^4 - 8\ z^2 + 1, 16\ z^5 - 20\ z^3 + 5\ z\right]$$

```
                              Type:  List Polynomial Integer
```

The expression *chebyshevT*(n, z) evaluates to the n-th Chebyshev polynomial of the first kind.

```
chebyshevT(3, 5.0 + 6.0*%i)
```

$$-1675.0 + 918.0\ i$$

<div align="right">Type: Complex Float</div>

```
chebyshevT(3, 5.0::DoubleFloat)
```

$$485.0$$

<div align="right">Type: DoubleFloat</div>

The expression $chebyshevU(n, z)$ evaluates to the n-th Chebyshev polynomial of the second kind.

```
[chebyshevU(i, z) for i in 0..5]
```

$$\left[1, 2\ z, 4\ z^2 - 1, 8\ z^3 - 4\ z, 16\ z^4 - 12\ z^2 + 1, 32\ z^5 - 32\ z^3 + 6\ z\right]$$

<div align="right">Type: List Polynomial Integer</div>

```
chebyshevU(3, 0.2)
```

$$-0.736$$

<div align="right">Type: Float</div>

The expression $hermiteH(n, z)$ evaluates to the n-th Hermite polynomial.

```
[hermiteH(i, z) for i in 0..5]
```

$$\left[1, 2\ z, 4\ z^2 - 2, 8\ z^3 - 12\ z, 16\ z^4 - 48\ z^2 + 12, 32\ z^5 - 160\ z^3 + 120\ z\right]$$

<div align="right">Type: List Polynomial Integer</div>

```
hermiteH(100, 1.0)
```

$$-0.1448706729337934088E93$$

<div align="right">Type: Float</div>

The expression $laguerreL(n, z)$ evaluates to the n-th Laguerre polynomial.

```
[laguerreL(i, z) for i in 0..4]
```

$$\left[1, -z + 1, z^2 - 4\ z + 2, -z^3 + 9\ z^2 - 18\ z + 6, z^4 - 16\ z^3 + 72\ z^2 - 96\ z + 24\right]$$

<div align="right">Type: List Polynomial Integer</div>

```
laguerreL(4, 1.2)
```

$$-13.0944$$

<div align="right">Type: Float</div>

```
[laguerreL(j, 3, z) for j in 0..4]
```

$$\left[-z^3 + 9\ z^2 - 18\ z + 6, -3\ z^2 + 18\ z - 18, -6\ z + 18, -6, 0\right]$$

<div align="right">Type: List Polynomial Integer</div>

```
laguerreL(1, 3, 2.1)
```

$$6.57$$

<div align="right">Type: Float</div>

The expression $legendreP(n, z)$ evaluates to the n-th Legendre polynomial,

```
[legendreP(i,z) for i in 0..5]
```

$$\left[1, z, \frac{3}{2}\ z^2 - \frac{1}{2}, \frac{5}{2}\ z^3 - \frac{3}{2}\ z, \frac{35}{8}\ z^4 - \frac{15}{4}\ z^2 + \frac{3}{8}, \frac{63}{8}\ z^5 - \frac{35}{4}\ z^3 + \frac{15}{8}\ z\right]$$

<div align="right">Type: List Polynomial Fraction Integer</div>

```
legendreP(3, 3.0*%i)
```

$$-72.0\ i$$

<div align="right">Type: Complex Float</div>

Finally, three number-theoretic polynomial operations may be evaluated. The following operations are provided by the package `NumberTheoreticPolynomialFunctions`. .

bernoulliB: $(NonNegativeInteger, R) -> R$
$bernoulliB(n, z)$ is the n-th Bernoulli polynomial, $B_n(z)$. These are defined by

$$\frac{te^{zt}}{e^t - 1} = \sum_{n=0}^{\infty} B_n(z) \frac{t^n}{n!}.$$

eulerE: $(NonNegativeInteger, R) -> R$
$eulerE(n, z)$ is the n-th Euler polynomial, $E_n(z)$. These are defined by

$$\frac{2e^{zt}}{e^t + 1} = \sum_{n=0}^{\infty} E_n(z) \frac{t^n}{n!}.$$

cyclotomic: $(NonNegativeInteger, R) -> R$
$cyclotomic(n, z)$ is the n-th cyclotomic polynomial $\Phi_n(z)$. This is the polynomial whose roots are precisely the primitive n-th roots of unity. This polynomial has degree given by the Euler totient function $\phi(n)$.

The expression $bernoulliB(n, z)$ evaluates to the n-th Bernoulli polynomial.

```
bernoulliB(3, z)
```

$$z^3 - \frac{3}{2} z^2 + \frac{1}{2} z$$

                          Type: Polynomial Fraction Integer

```
bernoulliB(3, 0.7 + 0.4 * %i)
```

$$-0.138 - 0.116\ i$$

                          Type: Complex Float

The expression $eulerE(n, z)$ evaluates to the n-th Euler polynomial.

```
eulerE(3, z)
```

$$z^3 - \frac{3}{2} z^2 + \frac{1}{4}$$

                          Type: Polynomial Fraction Integer

```
eulerE(3, 0.7 + 0.4 * %i)
```

$$-0.238 - 0.316\ i$$

<div align="right">Type: Complex Float</div>

The expression *cyclotomic*(*n*, *z*) evaluates to the *n*-th cyclotomic polynomial.

```
cyclotomic(3, z)
```

$$z^2 + z + 1$$

<div align="right">Type: Polynomial Integer</div>

```
cyclotomic(3, (-1.0 + 0.0 * %i)**(2/3))
```

$$0.0$$

<div align="right">Type: Complex Float</div>

Drawing complex functions in Axiom is presently somewhat awkward compared to drawing real functions. It is necessary to use the **draw** operations that operate on functions rather than expressions.

This is the complex exponential function (rotated interactively). When this is displayed in color, the height is the value of the real part of the function and the color is the imaginary part. Red indicates large negative imaginary values, green indicates imaginary values near zero and blue/violet indicates large positive imaginary values.

```
draw((x,y)+-> real exp complex(x,y), -2..2, -2*%pi..2*%pi,
colorFunction == (x, y) +-> imag exp complex(x,y),
title=="exp(x+%i*y)", style=="smooth")
```

This is the complex arctangent function. Again, the height is the real part of the function value but here the color indicates the function value's phase. The position of the branch cuts are clearly visible and one can see that the function is real only for a real argument.

```
vp := draw((x,y) +-> real atan complex(x,y), -%pi..%pi,
-%pi..%pi, colorFunction==(x,y) +->argument atan complex(x,y),
title=="atan(x+%i*y)", style=="shade"); rotate(vp,-160,-45); vp
```

This is the complex Gamma function.

```
draw((x,y) +-> max(min(real Gamma complex(x,y),4),-4), -%pi..%pi,
-%pi..%pi, style=="shade", colorFunction == (x,y) +-> argument
Gamma complex(x,y), title == "Gamma(x+%i*y)", var1Steps == 50,
var2Steps== 50)
```

This shows the real Beta function near the origin.

```
draw(Beta(x,y)/100, x=-1.6..1.7, y = -1.6..1.7, style=="shade",
title=="Beta(x,y)", var1Steps==40, var2Steps==40)
```

This is the Bessel function $J_\alpha(x)$ for index α in the range $-6..4$ and argument x in the range $2..14$.

```
draw((alpha,x) +-> min(max(besselJ(alpha, x+8), -6), 6), -6..4,
-6..6, title=="besselJ(alpha,x)", style=="shade", var1Steps==40,
var2Steps==40)
```

This is the modified Bessel function $I_\alpha(x)$ evaluated for various real values of the index α and fixed argument $x = 5$.

```
draw(besselI(alpha, 5), alpha = -12..12, unit==[5,20])
```

This is similar to the last example except the index α takes on complex values in a $6x6$ rectangle centered on the origin.

```
draw((x,y) +-> real besselI(complex(x/20, y/20),5), -60..60,
-60..60, colorFunction == (x,y)+-> argument
besselI(complex(x/20,y/20),5), title=="besselI(x+i*y,5)",
style=="shade")
```

8.2   Polynomial Factorization

The Axiom polynomial factorization facilities are available for all polynomial types and a wide variety of coefficient domains. Here are some examples.

8.2.1   Integer and Rational Number Coefficients

Polynomials with integer coefficients can be be factored.

```
v := (4*x**3+2*y**2+1)*(12*x**5-x**3*y+12)
```

$$-2\ x^3\ y^3 + \left(24\ x^5 + 24\right)\ y^2 + \left(-4\ x^6 - x^3\right)\ y + 48\ x^8 + 12\ x^5 + 48\ x^3 + 12$$

<div align="right">Type: Polynomial Integer</div>

```
factor v
```

$$-\left(x^3\ y - 12\ x^5 - 12\right)\left(2\ y^2 + 4\ x^3 + 1\right)$$

<div align="right">Type: Factored Polynomial Integer</div>

Also, Axiom can factor polynomials with rational number coefficients.

```
w := (4*x**3+(2/3)*x**2+1)*(12*x**5-(1/2)*x**3+12)
```

$$48\ x^8 + 8\ x^7 - 2\ x^6 + \frac{35}{3}\ x^5 + \frac{95}{2}\ x^3 + 8\ x^2 + 12$$

<div align="right">Type: Polynomial Fraction Integer</div>

```
factor w
```

$$48\left(x^3 + \frac{1}{6}\ x^2 + \frac{1}{4}\right)\left(x^5 - \frac{1}{24}\ x^3 + 1\right)$$

<div align="right">Type: Factored Polynomial Fraction Integer</div>

8.2.2 Finite Field Coefficients

Polynomials with coefficients in a finite field can be also be factored.

```
u :  POLY(PF(19)) :=3*x**4+2*x**2+15*x+18
```

$$3\ x^4 + 2\ x^2 + 15\ x + 18$$

<div align="right">Type: Polynomial PrimeField 19</div>

These include the integers mod p, where p is prime, and extensions of these fields.

```
factor u
```

$$3 \ (x + 18) \ (x^3 + x^2 + 8 \ x + 13)$$

<div align="right">Type: Factored Polynomial PrimeField 19</div>

Convert this to have coefficients in the finite field with 19^3 elements. See 8.11 on page 413 for more information about finite fields.

```
factor(u ::  POLY FFX(PF 19,3))
```

$$3 \ (x + 18) \ (x + 5 \ \%I^2 + 3 \ \%I + 13) \ (x + 16 \ \%I^2 + 14 \ \%I + 13) \ (x + 17 \ \%I^2 + 2 \ \%I + 13)$$

<div align="center">Type: Factored Polynomial FiniteFieldExtension(PrimeField 19,3)</div>

8.2.3   Simple Algebraic Extension Field Coefficients

Polynomials with coefficients in simple algebraic extensions of the rational numbers can be factored.

Here, *aa* and *bb* are symbolic roots of polynomials.

```
aa := rootOf(aa**2+aa+1)
```

$$aa$$

<div align="right">Type: AlgebraicNumber</div>

```
p:=(x**3+aa**2*x+y)*(aa*x**2+aa*x+aa*y**2)**2
```

$$(-aa - 1) \ y^5 + \left((-aa - 1) \ x^3 + aa \ x\right) y^4 +$$

$$\left((-2 \ aa - 2) \ x^2 + (-2 \ aa - 2) \ x\right) y^3 +$$

$$\left((-2 \ aa - 2) \ x^5 + (-2 \ aa - 2) \ x^4 + 2 \ aa \ x^3 + 2 \ aa \ x^2\right) y^2 +$$

$$\left((-aa - 1) \ x^4 + (-2 \ aa - 2) \ x^3 + (-aa - 1) \ x^2\right) y +$$

$$(-aa - 1) \ x^7 + (-2 \ aa - 2) \ x^6 - x^5 + 2 \ aa \ x^4 + aa \ x^3$$

<div align="right">Type: Polynomial AlgebraicNumber</div>

Note that the second argument to factor can be a list of algebraic extensions to factor over.

```
factor(p,[aa])
```

$$(-aa - 1)\ \left(y + x^3 + (-aa - 1)\ x\right)\ \left(y^2 + x^2 + x\right)^2$$

<div align="center">Type: Factored Polynomial AlgebraicNumber</div>

This factors $x * *2 + 3$ over the integers.

```
factor(x**2+3)
```

$$x^2 + 3$$

<div align="center">Type: Factored Polynomial Integer</div>

Factor the same polynomial over the field obtained by adjoining aa to the rational numbers.

```
factor(x**2+3,[aa])
```

$$(x - 2\ aa - 1)\ (x + 2\ aa + 1)$$

<div align="center">Type: Factored Polynomial AlgebraicNumber</div>

Factor $x * *6 + 108$ over the same field.

```
factor(x**6+108,[aa])
```

$$\left(x^3 - 12\ aa - 6\right)\ \left(x^3 + 12\ aa + 6\right)$$

<div align="center">Type: Factored Polynomial AlgebraicNumber</div>

```
bb:=rootOf(bb**3-2)
```

$$bb$$

<div align="center">Type: AlgebraicNumber</div>

```
factor(x**6+108,[bb])
```

$$\left(x^2 - 3\ bb\ x + 3\ bb^2\right)\left(x^2 + 3\ bb^2\right)\left(x^2 + 3\ bb\ x + 3\ bb^2\right)$$

<div align="center">Type: Factored Polynomial AlgebraicNumber</div>

Factor again over the field obtained by adjoining both *aa* and *bb* to the rational numbers.

```
factor(x**6+108,[aa,bb])
```

$$(x + (-2\ aa - 1)\ bb)\ (x + (-aa - 2)\ bb)\ (x + (-aa + 1)\ bb)$$

$$(x + (aa - 1)\ bb)\ (x + (aa + 2)\ bb)\ (x + (2\ aa + 1)\ bb)$$

<div align="center">Type: Factored Polynomial AlgebraicNumber</div>

8.2.4   Factoring Rational Functions

Since fractions of polynomials form a field, every element (other than zero) divides any other, so there is no useful notion of irreducible factors. Thus the **factor** operation is not very useful for fractions of polynomials.

There is, instead, a specific operation **factorFraction** that separately factors the numerator and denominator and returns a fraction of the factored results.

```
factorFraction((x**2-4)/(y**2-4))
```

$$\frac{(x - 2)\ (x + 2)}{(y - 2)\ (y + 2)}$$

<div align="center">Type: Fraction Factored Polynomial Integer</div>

You can also use **map**. This expression applies the **factor** operation to the numerator and denominator.

```
map(factor,(x**2-4)/(y**2-4))
```

$$\frac{(x - 2)\ (x + 2)}{(y - 2)\ (y + 2)}$$

<div align="center">Type: Fraction Factored Polynomial Integer</div>

8.3 Manipulating Symbolic Roots of a Polynomial

In this section we show you how to work with one root or all roots of a polynomial. These roots are represented symbolically (as opposed to being numeric approximations). See 8.5.2 on page 369 and 8.5.3 on page 371 for information about solving for the roots of one or more polynomials.

8.3.1 Using a Single Root of a Polynomial

Use **rootOf** to get a symbolic root of a polynomial: $rootOf(p, x)$ returns a root of $p(x)$.

This creates an algebraic number a.

```
a := rootOf(a**4+1,a)
```

$$a$$

Type: Expression Integer

To find the algebraic relation that defines a, use **definingPolynomial**.

```
definingPolynomial a
```

$$a^4 + 1$$

Type: Expression Integer

You can use a in any further expression, including a nested **rootOf**.

```
b := rootOf(b**2-a-1,b)
```

$$b$$

Type: Expression Integer

Higher powers of the roots are automatically reduced during calculations.

```
a + b
```

$$b + a$$

<div align="right">Type: Expression Integer</div>

```
% ** 5
```

$$\left(10\ a^3 + 11\ a^2 + 2\ a - 4\right)\ b + 15\ a^3 + 10\ a^2 + 4\ a - 10$$

<div align="right">Type: Expression Integer</div>

The operation **zeroOf** is similar to **rootOf**, except that it may express the root using radicals in some cases.

```
rootOf(c**2+c+1,c)
```

$$c$$

<div align="right">Type: Expression Integer</div>

```
zeroOf(d**2+d+1,d)
```

$$\frac{\sqrt{-3} - 1}{2}$$

<div align="right">Type: Expression Integer</div>

```
rootOf(e**5-2,e)
```

$$e$$

<div align="right">Type: Expression Integer</div>

```
zeroOf(f**5-2,f)
```

$$\sqrt[5]{2}$$

<div align="right">Type: Expression Integer</div>

8.3.2 Using All Roots of a Polynomial

Use **rootsOf** to get all symbolic roots of a polynomial: $rootsOf(p, x)$ returns a list of all the roots of $p(x)$. If $p(x)$ has a multiple root of order n, then that root appears n times in the list.

Compute all the roots of $x * *4 + 1$.

```
l := rootsOf(x**4+1,x)
```

$$[\%x0, \%x0\ \%x1, -\%x0, -\%x0\ \%x1]$$

<div align="right">Type: List Expression Integer</div>

As a side effect, the variables $\%x0, \%x1$ and $\%x2$ are bound to the first three roots of $x * *4 + 1$.

```
%x0**5
```

$$-\%x0$$

<div align="right">Type: Expression Integer</div>

Although they all satisfy $x * *4 + 1 = 0, \%x0, \%x1$, and $\%x2$ are different algebraic numbers. To find the algebraic relation that defines each of them, use **definingPolynomial**.

```
definingPolynomial %x0
```

$$\%x0^4 + 1$$

<div align="right">Type: Expression Integer</div>

```
definingPolynomial %x1
```

$$\%x1^2 + 1$$

<div align="right">Type: Expression Integer</div>

```
definingPolynomial %x2
```

$$-\%x2 + \%\%var$$

```
                                           Type:  Expression Integer
```

We can check that the sum and product of the roots of $x**4 + 1$ are its trace and norm.

```
x3 := last l
```

$$-\%x0 \; \%x1$$

```
                                           Type:  Expression Integer
```

```
%x0 + %x1 + %x2 + x3
```

$$(-\%x0 + 1) \; \%x1 + \%x0 + \%x2$$

```
                                           Type:  Expression Integer
```

```
%x0 * %x1 * %x2 * x3
```

$$\%x2 \; \%x0^2$$

```
                                           Type:  Expression Integer
```

Corresponding to the pair of operations **rootOf**/**zeroOf** in 8.5.2 on page 369, there is an operation **zerosOf** that, like **rootsOf**, computes all the roots of a given polynomial, but which expresses some of them in terms of radicals.

```
zerosOf(y**4+1,y)
```

$$\left[\frac{\sqrt{-1}+1}{\sqrt{2}}, \frac{\sqrt{-1}-1}{\sqrt{2}}, \frac{-\sqrt{-1}-1}{\sqrt{2}}, \frac{-\sqrt{-1}+1}{\sqrt{2}} \right]$$

```
                                       Type:  List Expression Integer
```

As you see, only one implicit algebraic number was created ($\%y1$), and its defining equation is this. The other three roots are expressed in radicals.

```
definingPolynomial %y1
```

$$\%\%var^2 + 1$$

```
                                           Type:  Expression Integer
```

8.4   Computation of Eigenvalues and Eigenvectors

In this section we show you some of Axiom's facilities for computing and manipulating eigenvalues and eigenvectors, also called characteristic values and characteristic vectors, respectively.

Let's first create a matrix with integer entries.

```
m1 := matrix [ [1,2,1],[2,1,-2],[1,-2,4] ]
```

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 1 & -2 \\ 1 & -2 & 4 \end{bmatrix}$$

            Type: Matrix Integer

To get a list of the *rational* eigenvalues, use the operation **eigenvalues**.

```
leig := eigenvalues(m1)
```

$$\left[5, \left(\%K \mid \%K^2 - \%K - 5 \right) \right]$$

         Type: List Union(Fraction Polynomial
Integer,SuchThat(Symbol,Polynomial Integer))

Given an explicit eigenvalue, **eigenvector** computes the eigenvectors corresponding to it.

```
eigenvector(first(leig),m1)
```

$$\left[\begin{bmatrix} 0 \\ -\frac{1}{2} \\ 1 \end{bmatrix} \right]$$

       Type: List Matrix Fraction Polynomial Fraction Integer

The operation **eigenvectors** returns a list of pairs of values and vectors. When an eigenvalue is rational, Axiom gives you the value explicitly; otherwise, its

minimal polynomial is given, (the polynomial of lowest degree with the eigen-
values as roots), together with a parametric representation of the eigenvector
using the eigenvalue. This means that if you ask Axiom to **solve** the minimal
polynomial, then you can substitute these roots into the parametric form of the
corresponding eigenvectors.

You must be aware that unless an exact eigenvalue has been computed, the
eigenvector may be badly in error.

`eigenvectors(m1)`

$$\left[\left[eigval = 5, eigmult = 1, eigvec = \left[\begin{bmatrix} 0 \\ -\frac{1}{2} \\ 1 \end{bmatrix}\right]\right],\right.$$

$$\left.\left[eigval = \left(\%L \mid \%L^2 - \%L - 5\right), eigmult = 1, eigvec = \left[\begin{bmatrix} \%L \\ 2 \\ 1 \end{bmatrix}\right]\right]\right]$$

Type: List Record(eigval: Union(Fraction Polynomial
Integer,SuchThat(Symbol,Polynomial Integer)),eigmult:
NonNegativeInteger,eigvec: List Matrix Fraction Polynomial
Integer)

Another possibility is to use the operation **radicalEigenvectors** tries to com-
pute explicitly the eigenvectors in terms of radicals.

`radicalEigenvectors(m1)`

$$\left[\left[radval = \frac{\sqrt{21}+1}{2}, radmult = 1, radvect = \left[\begin{bmatrix} \frac{\sqrt{21}+1}{2} \\ 2 \\ 1 \end{bmatrix}\right]\right],\right.$$

$$\left[radval = \frac{-\sqrt{21}+1}{2}, radmult = 1, radvect = \left[\begin{bmatrix} \frac{-\sqrt{21}+1}{2} \\ 2 \\ 1 \end{bmatrix}\right]\right],$$

$$\left.\left[radval = 5, radmult = 1, radvect = \left[\begin{bmatrix} 0 \\ -\frac{1}{2} \\ 1 \end{bmatrix}\right]\right]\right]$$

Type: List Record(radval: Expression Integer,radmult:
Integer,radvect: List Matrix Expression Integer)

Alternatively, Axiom can compute real or complex approximations to the eigenvectors and eigenvalues using the operations **realEigenvectors** or **complex-Eigenvectors**. They each take an additional argument ϵ to specify the "precision" required. In the real case, this means that each approximation will be within $\pm\epsilon$ of the actual result. In the complex case, this means that each approximation will be within $\pm\epsilon$ of the actual result in each of the real and imaginary parts.

The precision can be specified as a `Float` if the results are desired in floating-point notation, or as `Fraction Integer` if the results are to be expressed using rational (or complex rational) numbers.

`realEigenvectors(m1,1/1000)`

$$\left[\left[outval = 5, outmult = 1, outvect = \left[\left[\begin{array}{c} 0 \\ -\frac{1}{2} \\ 1 \end{array}\right]\right]\right],\right.$$

$$\left[outval = \frac{5717}{2048}, outmult = 1, outvect = \left[\left[\begin{array}{c} \frac{5717}{2048} \\ 2 \\ 1 \end{array}\right]\right]\right],$$

$$\left.\left[outval = -\frac{3669}{2048}, outmult = 1, outvect = \left[\left[\begin{array}{c} -\frac{3669}{2048} \\ 2 \\ 1 \end{array}\right]\right]\right]\right]$$

> Type: List Record(outval: Fraction Integer,outmult: Integer,outvect: List Matrix Fraction Integer)

If an n by n matrix has n distinct eigenvalues (and therefore n eigenvectors) the operation **eigenMatrix** gives you a matrix of the eigenvectors.

`eigenMatrix(m1)`

$$\left[\begin{array}{ccc} \frac{\frac{\sqrt{21}+1}{2}}{2} & \frac{\frac{-\sqrt{21}+1}{2}}{2} & 0 \\ 2 & 2 & -\frac{1}{2} \\ 1 & 1 & 1 \end{array}\right]$$

> Type: Union(Matrix Expression Integer,...)

`m2 := matrix [ [-5,-2],[18,7] ]`

$$\left[\begin{array}{cc} -5 & -2 \\ 18 & 7 \end{array}\right]$$

```
                                        Type:  Matrix Integer
```

```
eigenMatrix(m2)
```

```
                        "failed"
```

```
                                Type:  Union("failed",...)
```

If a symmetric matrix has a basis of orthonormal eigenvectors, then **orthonormalBasis** computes a list of these vectors.

```
m3 := matrix [ [1,2],[2,1] ]
```

$$\begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$$

```
                                        Type:  Matrix Integer
```

```
orthonormalBasis(m3)
```

$$\left[\left[\begin{array}{c} -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{array} \right], \left[\begin{array}{c} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{array} \right] \right]$$

```
                        Type:  List Matrix Expression Integer
```

8.5 Solution of Linear and Polynomial Equations

In this section we discuss the Axiom facilities for solving systems of linear equations, finding the roots of polynomials and solving systems of polynomial equations. For a discussion of the solution of differential equations, see 8.10 on page 402.

8.5.1 Solution of Systems of Linear Equations

You can use the operation **solve** to solve systems of linear equations.

The operation **solve** takes two arguments, the list of equations and the list of the unknowns to be solved for. A system of linear equations need not have a unique solution.

To solve the linear system:

$$
\begin{array}{rcrcrcr}
x & + & y & + & z & = & 8 \\
3x & - & 2y & + & z & = & 0 \\
x & + & 2y & + & 2z & = & 17
\end{array}
$$

evaluate this expression.

```
solve([x+y+z=8,3*x-2*y+z=0,x+2*y+2*z=17],[x,y,z])
```

$$[[x = -1, y = 2, z = 7]]$$

Type: List List Equation Fraction Polynomial Integer

Parameters are given as new variables starting with a percent sign and % and the variables are expressed in terms of the parameters. If the system has no solutions then the empty list is returned.

When you solve the linear system

$$
\begin{array}{rcrcrcr}
x & + & 2y & + & 3z & = & 2 \\
2x & + & 3y & + & 4z & = & 2 \\
3x & + & 4y & + & 5z & = & 2
\end{array}
$$

with this expression you get a solution involving a parameter.

```
solve([x+2*y+3*z=2,2*x+3*y+4*z=2,3*x+4*y+5*z=2],[x,y,z])
```

$$[[x = \%Q - 2, y = -2\ \%Q + 2, z = \%Q]]$$

Type: List List Equation Fraction Polynomial Integer

The system can also be presented as a matrix and a vector. The matrix contains the coefficients of the linear equations and the vector contains the numbers appearing on the right-hand sides of the equations. You may input the matrix as a list of rows and the vector as a list of its elements.

To solve the system:

$$
\begin{array}{rcrcrcr}
x & + & y & + & z & = & 8 \\
3x & - & 2y & + & z & = & 0 \\
x & + & 2y & + & 2z & = & 17
\end{array}
$$

in matrix form you would evaluate this expression.

```
solve([ [1,1,1],[3,-2,1],[1,2,2] ],[8,0,17])
```

$$[particular = [-1, 2, 7], basis = [[0, 0, 0]]]$$

```
          Type:  Record(particular:  Union(Vector Fraction
Integer,"failed"), basis:  List Vector Fraction Integer)
```

The solutions are presented as a `Record` with two components: the compo-
nent *particular* contains a particular solution of the given system or the item
`"failed"` if there are no solutions, the component *basis* contains a list of vec-
tors that are a basis for the space of solutions of the corresponding homogeneous
system. If the system of linear equations does not have a unique solution, then
the *basis* component contains non-trivial vectors.

This happens when you solve the linear system

$$
\begin{array}{rcrcrcl}
x & + & 2y & + & 3z & = & 2 \\
2x & + & 3y & + & 4z & = & 2 \\
3x & + & 4y & + & 5z & = & 2
\end{array}
$$

with this command.

```
solve([ [1,2,3],[2,3,4],[3,4,5] ],[2,2,2])
```

$$[particular = [-2, 2, 0], basis = [[1, -2, 1]]]$$

```
          Type:  Record(particular:  Union(Vector Fraction
Integer,"failed"), basis:  List Vector Fraction Integer)
```

All solutions of this system are obtained by adding the particular solution with
a linear combination of the *basis* vectors.

When no solution exists then `"failed"` is returned as the *particular* component,
as follows:

```
solve([ [1,2,3],[2,3,4],[3,4,5] ],[2,3,2])
```

$$[particular = \texttt{"failed"}, basis = [[1, -2, 1]]]$$

```
          Type:  Record(particular:  Union(Vector Fraction
Integer,"failed"), basis:  List Vector Fraction Integer)
```

When you want to solve a system of homogeneous equations (that is, a system
where the numbers on the right-hand sides of the equations are all zero) in
the matrix form you can omit the second argument and use the **nullSpace**
operation.

This computes the solutions of the following system of equations:

$$\begin{array}{ccccccc} x & + & 2y & + & 3z & = & 0 \\ 2x & + & 3y & + & 4z & = & 0 \\ 3x & + & 4y & + & 5z & = & 0 \end{array}$$

The result is given as a list of vectors and these vectors form a basis for the solution space.

```
nullSpace([ [1,2,3],[2,3,4],[3,4,5] ])
```

$$[[1, -2, 1]]$$

<div align="right">Type: List Vector Integer</div>

8.5.2 Solution of a Single Polynomial Equation

Axiom can solve polynomial equations producing either approximate or exact solutions. Exact solutions are either members of the ground field or can be presented symbolically as roots of irreducible polynomials.

This returns the one rational root along with an irreducible polynomial describing the other solutions.

```
solve(x**3 = 8,x)
```

$$\left[x = 2, x^2 + 2\ x + 4 = 0\right]$$

<div align="right">Type: List Equation Fraction Polynomial Integer</div>

If you want solutions expressed in terms of radicals you would use this instead.

```
radicalSolve(x**3 = 8,x)
```

$$\left[x = -\sqrt{-3} - 1, x = \sqrt{-3} - 1, x = 2\right]$$

<div align="right">Type: List Equation Expression Integer</div>

The **solve** command always returns a value but **radicalSolve** returns only the solutions that it is able to express in terms of radicals.

If the polynomial equation has rational coefficients you can ask for approximations to its real roots by calling solve with a second argument that specifies the "precision" ϵ. This means that each approximation will be within $\pm\epsilon$ of the actual result.

Notice that the type of second argument controls the type of the result.

```
solve(x**4 - 10*x**3 + 35*x**2 - 50*x + 25,.0001)
```

$$[x = 3.618011474609375, x = 1.381988525390625]$$

<div align="right">Type: List Equation Polynomial Float</div>

If you give a floating-point precision you get a floating-point result; if you give the precision as a rational number you get a rational result.

```
solve(x**3-2,1/1000)
```

$$\left[x = \frac{2581}{2048} \right]$$

<div align="center">Type: List Equation Polynomial Fraction Integer</div>

If you want approximate complex results you should use the command **complexSolve** that takes the same precision argument ϵ.

```
complexSolve(x**3-2,.0001)
```

$$[x = 1.259918212890625,$$

$$x = -0.62989432795395613131 - 1.091094970703125 \; i,$$

$$x = -0.62989432795395613131 + 1.091094970703125 \; i]$$

<div align="right">Type: List Equation Polynomial Complex Float</div>

Each approximation will be within $\pm\epsilon$ of the actual result in each of the real and imaginary parts.

```
complexSolve(x**2-2*%i+1,1/100)
```

$$\left[x = -\frac{13028925}{16777216} - \frac{325}{256} \; i, x = \frac{13028925}{16777216} + \frac{325}{256} \; i \right]$$

<div align="center">Type: List Equation Polynomial Complex Fraction Integer</div>

Note that if you omit the **=** from the first argument Axiom generates an equation by equating the first argument to zero. Also, when only one variable is present in the equation, you do not need to specify the variable to be solved for, that is, you can omit the second argument.

Axiom can also solve equations involving rational functions. Solutions where the denominator vanishes are discarded.

```
radicalSolve(1/x**3 + 1/x**2 + 1/x = 0,x)
```

$$\left[x = \frac{-\sqrt{-3}-1}{2}, x = \frac{\sqrt{-3}-1}{2}\right]$$

<div align="center">Type: List Equation Expression Integer</div>

8.5.3 Solution of Systems of Polynomial Equations

Given a system of equations of rational functions with exact coefficients:

$$p_1(x_1,\ldots,x_n)$$
$$\vdots$$
$$p_m(x_1,\ldots,x_n)$$

Axiom can find numeric or symbolic solutions. The system is first split into irreducible components, then for each component, a triangular system of equations is found that reduces the problem to sequential solution of univariate polynomials resulting from substitution of partial solutions from the previous stage.

$$q_1(x_1,\ldots,x_n)$$
$$\vdots$$
$$q_m(x_n)$$

Symbolic solutions can be presented using "implicit" algebraic numbers defined as roots of irreducible polynomials or in terms of radicals. Axiom can also find approximations to the real or complex roots of a system of polynomial equations to any user-specified accuracy.

The operation **solve** for systems is used in a way similar to **solve** for single equations. Instead of a polynomial equation, one has to give a list of equations and instead of a single variable to solve for, a list of variables. For solutions of single equations see 8.5.2 on page 369.

Use the operation **solve** if you want implicitly presented solutions.

```
solve([3*x**3 + y + 1,y**2 -4],[x,y])
```

$$\left[[x = -1, y = 2], \left[x^2 - x + 1 = 0, y = 2\right], \left[3\ x^3 - 1 = 0, y = -2\right]\right]$$

<div align="center">Type: List List Equation Fraction Polynomial Integer</div>

```
solve([x = y**2-19,y = z**2+x+3,z = 3*x],[x,y,z])
```

$$\left[\left[x = \frac{z}{3}, y = \frac{3\ z^2 + z + 9}{3}, 9\ z^4 + 6\ z^3 + 55\ z^2 + 15\ z - 90 = 0\right]\right]$$

<div align="center">Type: List List Equation Fraction Polynomial Integer</div>

Use **radicalSolve** if you want your solutions expressed in terms of radicals.

```
radicalSolve([3*x**3 + y + 1,y**2 -4],[x,y])
```

$$\left[\left[x = \frac{\sqrt{-3}+1}{2}, y = 2\right], \left[x = \frac{-\sqrt{-3}+1}{2}, y = 2\right],\right.$$

$$\left[x = \frac{-\sqrt{-1}\ \sqrt{3} - 1}{2\ \sqrt[3]{3}}, y = -2\right], \left[x = \frac{\sqrt{-1}\ \sqrt{3} - 1}{2\ \sqrt[3]{3}}, y = -2\right],$$

$$\left.\left[x = \frac{1}{\sqrt[3]{3}}, y = -2\right], [x = -1, y = 2]\right]$$

<div align="center">Type: List List Equation Expression Integer</div>

To get numeric solutions you only need to give the list of equations and the precision desired. The list of variables would be redundant information since there can be no parameters for the numerical solver.

If the precision is expressed as a floating-point number you get results expressed as floats.

```
solve([x**2*y - 1,x*y**2 - 2],.01)
```

$$[[y = 1.5859375, x = 0.79296875]]$$

<div align="center">Type: List List Equation Polynomial Float</div>

To get complex numeric solutions, use the operation **complexSolve**, which takes the same arguments as in the real case.

```
complexSolve([x**2*y - 1,x*y**2 - 2],1/1000)
```

$$\left[\left[y = \tfrac{1625}{1024}, x = \tfrac{1625}{2048}\right],\right.$$

$$\left[y = -\frac{435445573689}{549755813888} - \frac{1407}{1024}\ i, x = -\frac{435445573689}{1099511627776} - \frac{1407}{2048}\ i\right],$$

$$\left.\left[y = -\frac{435445573689}{549755813888} + \frac{1407}{1024}\ i, x = -\frac{435445573689}{1099511627776} + \frac{1407}{2048}\ i\right]\right]$$

        Type: List List Equation Polynomial Complex Fraction Integer

It is also possible to solve systems of equations in rational functions over the
rational numbers. Note that $[x = 0.0, a = 0.0]$ is not returned as a solution
since the denominator vanishes there.

```
solve([x**2/a = a,a = a*x],.001)
```

$$[[x = 1.0, a = -1.0], [x = 1.0, a = 1.0]]$$

                        Type: List List Equation Polynomial Float

When solving equations with denominators, all solutions where the denominator
vanishes are discarded.

```
radicalSolve([x**2/a + a + y**3 - 1,a*y + a + 1],[x,y])
```

$$\left[\left[x = -\sqrt{\frac{-a^4+2\ a^3+3\ a^2+3\ a+1}{a^2}}, y = \frac{-a-1}{a}\right],\right.$$

$$\left.\left[x = \sqrt{\frac{-a^4 + 2\ a^3 + 3\ a^2 + 3\ a + 1}{a^2}}, y = \frac{-a-1}{a}\right]\right]$$

                Type: List List Equation Expression Integer

8.6   Limits

To compute a limit, you must specify a functional expression, a variable, and
a limiting value for that variable. If you do not specify a direction, Axiom
attempts to compute a two-sided limit.

Issue this to compute the limit

$$\lim_{x \to 1} \frac{x^2 - 3x + 2}{x^2 - 1}.$$

```
limit((x**2 - 3*x + 2)/(x**2 - 1),x = 1)
```

$$-\frac{1}{2}$$

   Type: Union(OrderedCompletion Fraction Polynomial Integer,...)

Sometimes the limit when approached from the left is different from the limit
from the right and, in this case, you may wish to ask for a one-sided limit. Also,
if you have a function that is only defined on one side of a particular value, you
can compute a one-sided limit.

The function $log(x)$ is only defined to the right of zero, that is, for $x > 0$.
Thus, when computing limits of functions involving $log(x)$, you probably want
a "right-hand" limit.

```
limit(x * log(x),x = 0,"right")
```

$$0$$

```
        Type:  Union(OrderedCompletion Expression Integer,...)
```

When you do not specify "*right*" or "*left*" as the optional fourth argument,
limit tries to compute a two-sided limit. Here the limit from the left does not
exist, as Axiom indicates when you try to take a two-sided limit.

```
limit(x * log(x),x = 0)
```

$$[leftHandLimit = \texttt{"failed"}, rightHandLimit = 0]$$

```
    Type:  Union(Record(leftHandLimit:  Union(OrderedCompletion
                  Expression Integer,"failed"), rightHandLimit:
        Union(OrderedCompletion Expression Integer,"failed")),...)
```

A function can be defined on both sides of a particular value, but tend to
different limits as its variable approaches that value from the left and from the
right. We can construct an example of this as follows: Since $\sqrt{y^2}$ is simply the
absolute value of y, the function $\sqrt{y^2}/y$ is simply the sign $(+1$ or $-1)$ of the
nonzero real number y. Therefore, $\sqrt{y^2}/y = -1$ for $y < 0$ and $\sqrt{y^2}/y = +1$ for
$y > 0$.

This is what happens when we take the limit at $y = 0$. The answer returned by
Axiom gives both a "left-hand" and a "right-hand" limit.

```
limit(sqrt(y**2)/y,y = 0)
```

$$[leftHandLimit = -1, rightHandLimit = 1]$$

```
    Type:  Union(Record(leftHandLimit:  Union(OrderedCompletion
                  Expression Integer,"failed"), rightHandLimit:
        Union(OrderedCompletion Expression Integer,"failed")),...)
```

Here is another example, this time using a more complicated function.

```
limit(sqrt(1 - cos(t))/t,t = 0)
```

$$\left[leftHandLimit = -\frac{1}{\sqrt{2}}, rightHandLimit = \frac{1}{\sqrt{2}} \right]$$

> Type: Union(Record(leftHandLimit: Union(OrderedCompletion
> Expression Integer,"failed"), rightHandLimit:
> Union(OrderedCompletion Expression Integer,"failed")),...)

You can compute limits at infinity by passing either $+\infty$ or $-\infty$ as the third argument of **limit**.

To do this, use the constants *%plusInfinity* and *%minusInfinity*.

```
limit(sqrt(3*x**2 + 1)/(5*x),x = %plusInfinity)
```

$$\frac{\sqrt{3}}{5}$$

> Type: Union(OrderedCompletion Expression Integer,...)

```
limit(sqrt(3*x**2 + 1)/(5*x),x = %minusInfinity)
```

$$-\frac{\sqrt{3}}{5}$$

> Type: Union(OrderedCompletion Expression Integer,...)

You can take limits of functions with parameters. As you can see, the limit is expressed in terms of the parameters.

```
limit(sinh(a*x)/tan(b*x),x = 0)
```

$$\frac{a}{b}$$

> Type: Union(OrderedCompletion Expression Integer,...)

When you use **limit**, you are taking the limit of a real function of a real variable.

When you compute this, Axiom returns 0 because, as a function of a real variable, $sin(1/z)$ is always between -1 and 1, so $z * sin(1/z)$ tends to 0 as z tends to 0.

```
limit(z * sin(1/z),z = 0)
```

$$0$$

                    Type: Union(OrderedCompletion Expression Integer,...)

However, as a function of a *complex* variable, $sin(1/z)$ is badly behaved near 0 (one says that $sin(1/z)$ has an *essential singularity* at $z = 0$).

When viewed as a function of a complex variable, $z*sin(1/z)$ does not approach any limit as z tends to 0 in the complex plane. Axiom indicates this when we call **complexLimit**.

```
complexLimit(z * sin(1/z),z = 0)
```

                              "failed"

                                    Type: Union("failed",...)

Here is another example. As x approaches 0 along the real axis, $exp(-1/x**2)$ tends to 0.

```
limit(exp(-1/x**2),x = 0)
```

$$0$$

                    Type: Union(OrderedCompletion Expression Integer,...)

However, if x is allowed to approach 0 along any path in the complex plane, the limiting value of $exp(-1/x**2)$ depends on the path taken because the function has an essential singularity at $x = 0$. This is reflected in the error message returned by the function.

```
complexLimit(exp(-1/x**2),x = 0)
```

                              "failed"

                                    Type: Union("failed",...)

You can also take complex limits at infinity, that is, limits of a function of z as z approaches infinity on the Riemann sphere. Use the symbol $\%infinity$ to denote "complex infinity."

As above, to compute complex limits rather than real limits, use **complexLimit**.

```
complexLimit((2 + z)/(1 - z),z = %infinity)
```

$$-1$$

          Type: OnePointCompletion Fraction Polynomial Integer

In many cases, a limit of a real function of a real variable exists when the corresponding complex limit does not. This limit exists.

```
limit(sin(x)/x,x = %plusInfinity)
```

$$0$$

          Type: Union(OrderedCompletion Expression Integer,...)

But this limit does not.

```
complexLimit(sin(x)/x,x = %infinity)
```

$$\text{"failed"}$$

                              Type: Union("failed",...)

8.7   Laplace Transforms

Axiom can compute some forward Laplace transforms, mostly of elementary functions not involving logarithms, although some cases of special functions are handled.

To compute the forward Laplace transform of $F(t)$ with respect to t and express the result as $f(s)$, issue the command $laplace(F(t), t, s)$.

```
laplace(sin(a*t)*cosh(a*t)-cos(a*t)*sinh(a*t), t, s)
```

$$\frac{4\ a^3}{s^4 + 4\ a^4}$$

                              Type: Expression Integer

Here are some other non-trivial examples.

```
laplace((exp(a*t) - exp(b*t))/t, t, s)
```

$$-\log{(s-a)} + \log{(s-b)}$$

<div align="right">Type: Expression Integer</div>

```
laplace(2/t * (1 - cos(a*t)), t, s)
```

$$\log{(s^2 + a^2)} - 2\,\log{(s)}$$

<div align="right">Type: Expression Integer</div>

```
laplace(exp(-a*t) * sin(b*t) / b**2, t, s)
```

$$\frac{1}{b\,s^2 + 2\,a\,b\,s + b^3 + a^2\,b}$$

<div align="right">Type: Expression Integer</div>

```
laplace((cos(a*t) - cos(b*t))/t, t, s)
```

$$\frac{\log{(s^2 + b^2)} - \log{(s^2 + a^2)}}{2}$$

<div align="right">Type: Expression Integer</div>

Axiom also knows about a few special functions.

```
laplace(exp(a*t+b)*Ei(c*t), t, s)
```

$$\frac{e^b\,\log{\left(\frac{s+c-a}{c}\right)}}{s-a}$$

<div align="right">Type: Expression Integer</div>

```
laplace(a*Ci(b*t) + c*Si(d*t), t, s)
```

$$\frac{a\,\log{\left(\frac{s^2+b^2}{b^2}\right)} + 2\,c\,\arctan{\left(\frac{d}{s}\right)}}{2\,s}$$

<div align="right">Type: Expression Integer</div>

When Axiom does not know about a particular transform, it keeps it as a formal transform in the answer.

```
laplace(sin(a*t) - a*t*cos(a*t) + exp(t**2), t, s)
```

$$\frac{\left(s^4 + 2\ a^2\ s^2 + a^4\right)\ laplace\left(e^{t^2}, t, s\right) + 2\ a^3}{s^4 + 2\ a^2\ s^2 + a^4}$$

<div align="right">Type: Expression Integer</div>

8.8   Integration

Integration is the reverse process of differentiation, that is, an *integral* of a function f with respect to a variable x is any function g such that $D(g, x)$ is equal to f.

The package `FunctionSpaceIntegration` provides the top-level integration operation, **integrate**, for integrating real-valued elementary functions.

```
integrate(cosh(a*x)*sinh(a*x), x)
```

$$\frac{\sinh\left(a\ x\right)^2 + \cosh\left(a\ x\right)^2}{4\ a}$$

<div align="right">Type: Union(Expression Integer,...)</div>

Unfortunately, antiderivatives of most functions cannot be expressed in terms of elementary functions.

```
integrate(log(1 + sqrt(a * x + b)) / x, x)
```

$$\int^x \frac{\log\left(\sqrt{b + \%M\ a} + 1\right)}{\%M}\ d\%M$$

<div align="right">Type: Union(Expression Integer,...)</div>

Given an elementary function to integrate, Axiom returns a formal integral as above only when it can prove that the integral is not elementary and not when it cannot determine the integral. In this rare case it prints a message that it cannot determine if an elementary integral exists.

Similar functions may have antiderivatives that look quite different because the form of the antiderivative depends on the sign of a constant that appears in the function.

```
integrate(1/(x**2 - 2),x)
```

$$\frac{\log\left(\frac{\left(x^2+2\right)\ \sqrt{2}-4\ x}{x^2-2}\right)}{2\ \sqrt{2}}$$

<div align="right">Type: Union(Expression Integer,...)</div>

`integrate(1/(x**2 + 2),x)`

$$\frac{\arctan\left(\frac{x\ \sqrt{2}}{2}\right)}{\sqrt{2}}$$

<div align="right">Type: Union(Expression Integer,...)</div>

If the integrand contains parameters, then there may be several possible antiderivatives, depending on the signs of expressions of the parameters.

In this case Axiom returns a list of answers that cover all the possible cases. Here you use the answer involving the square root of a when $a > 0$ and the answer involving the square root of $-a$ when $a < 0$.

`integrate(x**2 / (x**4 - a**2), x)`

$$\left[\frac{\log\left(\frac{\left(x^2+a\right)\ \sqrt{a}-2\ a\ x}{x^2-a}\right)+2\ \arctan\left(\frac{x\ \sqrt{a}}{a}\right)}{4\ \sqrt{a}},\right.$$

$$\left.\frac{\log\left(\frac{\left(x^2-a\right)\ \sqrt{-a}+2\ a\ x}{x^2+a}\right)-2\ \arctan\left(\frac{x\ \sqrt{-a}}{a}\right)}{4\ \sqrt{-a}}\right]$$

<div align="right">Type: Union(List Expression Integer,...)</div>

If the parameters and the variables of integration can be complex numbers rather than real, then the notion of sign is not defined. In this case all the possible answers can be expressed as one complex function. To get that function, rather than a list of real functions, use **complexIntegrate**, which is provided by the package `FunctionSpaceComplexIntegration`.

This operation is used for integrating complex-valued elementary functions.

`complexIntegrate(x**2 / (x**4 - a**2), x)`

$$\frac{\left(\begin{array}{c} \sqrt{4\ a}\ \log\left(\frac{x\ \sqrt{-4\ a}+2\ a}{\sqrt{-4\ a}}\right) - \sqrt{-4\ a}\ \log\left(\frac{x\ \sqrt{4\ a}+2\ a}{\sqrt{4\ a}}\right)+ \\[2ex] \sqrt{-4\ a}\ \log\left(\frac{x\ \sqrt{4\ a}-2\ a}{\sqrt{4\ a}}\right) - \sqrt{4\ a}\ \log\left(\frac{x\ \sqrt{-4\ a}-2\ a}{\sqrt{-4\ a}}\right) \end{array}\right)}{2\ \sqrt{-4\ a}\ \sqrt{4\ a}}$$

<div align="right">Type: Expression Integer</div>

As with the real case, antiderivatives for most complex-valued functions cannot be expressed in terms of elementary functions.

```
complexIntegrate(log(1 + sqrt(a * x + b)) / x, x)
```

$$\int^x \frac{\log\left(\sqrt{b+\%M\ a}+1\right)}{\%M}\ d\%M$$

<div align="right">Type: Expression Integer</div>

Sometimes **integrate** can involve symbolic algebraic numbers such as those returned by **rootOf**. To see how to work with these strange generated symbols (such as %%a0), see 8.3.2 on page 361.

Definite integration is the process of computing the area between the x-axis and the curve of a function $f(x)$. The fundamental theorem of calculus states that if f is continuous on an interval $a..b$ and if there exists a function g that is differentiable on $a..b$ and such that $D(g, x)$ is equal to f, then the definite integral of f for x in the interval $a..b$ is equal to $g(b) - g(a)$.

The package `RationalFunctionDefiniteIntegration` provides the top-level definite integration operation, **integrate**, for integrating real-valued rational functions.

```
integrate((x**4 - 3*x**2 + 6)/(x**6-5*x**4+5*x**2+4), x = 1..2)
```

$$\frac{2\arctan\left(8\right) + 2\arctan\left(5\right) + 2\arctan\left(2\right) + 2\arctan\left(\frac{1}{2}\right) - \pi}{2}$$

<div align="center">Type: Union(f1:   OrderedCompletion Expression Integer,...)</div>

Axiom checks beforehand that the function you are integrating is defined on the interval $a..b$, and prints an error message if it finds that this is not case, as in the following example:

```
integrate(1/(x**2-2), x = 1..2)
```

```
>> Error detected within library code:
   Pole in path of integration
   You are being returned to the top level
   of the interpreter.
```

When parameters are present in the function, the function may or may not be defined on the interval of integration.

If this is the case, Axiom issues a warning that a pole might lie in the path of integration, and does not compute the integral.

```
integrate(1/(x**2-a), x = 1..2)
```

$$potentialPole$$

<div align="center">Type: Union(pole: potentialPole,...)</div>

If you know that you are using values of the parameter for which the function has no pole in the interval of integration, use the string ``noPole'' as a third argument to **integrate**:

The value here is, of course, incorrect if $sqrt(a)$ is between 1 and 2.

```
integrate(1/(x**2-a), x = 1..2, "noPole")
```

$$
\left[\begin{array}{l}
\dfrac{\left(-\log\left(\dfrac{\left(-4\ a^2-4\ a\right)\ \sqrt{a}+a^3+6\ a^2+a}{a^2-2\ a+1}\right)+\right.}{\left.\log\left(\dfrac{\left(-8\ a^2-32\ a\right)\ \sqrt{a}+a^3+24\ a^2+16\ a}{a^2-8\ a+16}\right)\right)}{4\ \sqrt{a},} \\[4em]
\dfrac{-\arctan\left(\dfrac{2\ \sqrt{-a}}{a}\right)+\arctan\left(\dfrac{\sqrt{-a}}{a}\right)}{\sqrt{-a}}
\end{array}\right]
$$

<div align="center">Type: Union(f2: List OrderedCompletion Expression Integer,...)</div>

8.9 Working with Power Series

Axiom has very sophisticated facilities for working with power series.

Infinite series are represented by a list of the coefficients that have already been determined, together with a function for computing the additional coefficients if needed.

The system command that determines how many terms of a series is displayed is `)set streams calculate`. For the purposes of this book, we have used this system command to display fewer than ten terms. Series can be created from expressions, from functions for the series coefficients, and from applications of operations on existing series. The most general function for creating a series is called **series**, although you can also use **taylor**, **laurent** and **puiseux** in situations where you know what kind of exponents are involved.

For information about solving differential equations in terms of power series, see

8.9.1 Creation of Power Series

This is the easiest way to create a power series. This tells Axiom that x is to be treated as a power series, so functions of x are again power series.

```
x := series 'x
```

$$x$$

         Type: UnivariatePuiseuxSeries(Expression Integer,x,0)

We didn't say anything about the coefficients of the power series, so the coefficients are general expressions over the integers. This allows us to introduce denominators, symbolic constants, and other variables as needed.

Here the coefficients are integers (note that the coefficients are the Fibonacci numbers).

```
1/(1 - x - x**2)
```

$$1+x+2\ x^2+3\ x^3+5\ x^4+8\ x^5+13\ x^6+21\ x^7+34\ x^8+55\ x^9+89\ x^{10}+O\left(x^{11}\right)$$

         Type: UnivariatePuiseuxSeries(Expression Integer,x,0)

This series has coefficients that are rational numbers.

```
sin(x)
```

$$x - \frac{1}{6}\,x^3 + \frac{1}{120}\,x^5 - \frac{1}{5040}\,x^7 + \frac{1}{362880}\,x^9 - \frac{1}{39916800}\,x^{11} + O\left(x^{12}\right)$$

<div align="center">Type: UnivariatePuiseuxSeries(Expression Integer,x,0)</div>

When you enter this expression you introduce the symbolic constants $sin(1)$ and $cos(1)$.

```
sin(1 + x)
```

$$\sin(1) + \cos(1)\,x - \frac{\sin(1)}{2}\,x^2 - \frac{\cos(1)}{6}\,x^3 + \frac{\sin(1)}{24}\,x^4 + \frac{\cos(1)}{120}\,x^5 - \frac{\sin(1)}{720}\,x^6 -$$

$$\frac{\cos(1)}{5040}\,x^7 + \frac{\sin(1)}{40320}\,x^8 + \frac{\cos(1)}{362880}\,x^9 - \frac{\sin(1)}{3628800}\,x^{10} + O\left(x^{11}\right)$$

<div align="center">Type: UnivariatePuiseuxSeries(Expression Integer,x,0)</div>

When you enter the expression the variable a appears in the resulting series expansion.

```
sin(a * x)
```

$$a\,x - \frac{a^3}{6}\,x^3 + \frac{a^5}{120}\,x^5 - \frac{a^7}{5040}\,x^7 + \frac{a^9}{362880}\,x^9 - \frac{a^{11}}{39916800}\,x^{11} + O\left(x^{12}\right)$$

<div align="center">Type: UnivariatePuiseuxSeries(Expression Integer,x,0)</div>

You can also convert an expression into a series expansion. This expression creates the series expansion of $1/log(y)$ about $y = 1$. For details and more examples, see 8.9.5 on page 391.

```
series(1/log(y),y = 1)
```

$$(y-1)^{(-1)} + \tfrac{1}{2} - \tfrac{1}{12}\,(y-1) + \tfrac{1}{24}\,(y-1)^2 - \tfrac{19}{720}\,(y-1)^3 + \tfrac{3}{160}\,(y-1)^4 -$$

$$\frac{863}{60480}\,(y-1)^5 + \frac{275}{24192}\,(y-1)^6 - \frac{33953}{3628800}\,(y-1)^7 +$$

$$\frac{8183}{1036800}\,(y-1)^8 - \frac{3250433}{479001600}\,(y-1)^9 + O\left((y-1)^{10}\right)$$

<div align="center">Type: UnivariatePuiseuxSeries(Expression Integer,y,1)</div>

You can create power series with more general coefficients. You normally accomplish this via a type declaration (see 2.3 on page 142). See 8.9.4 on page 388 for some warnings about working with declared series.

We declare that y is a one-variable Taylor series (`UTS` is the abbreviation for `UnivariateTaylorSeries`) in the variable z with `FLOAT` (that is, floating-point) coefficients, centered about 0. Then, by assignment, we obtain the Taylor expansion of $exp(z)$ with floating-point coefficients.

```
y :  UTS(FLOAT,'z,0) := exp(z)
```

$1.0 + z + 0.5\ z^2 + 0.1666666666\ 6666666667\ z^3 +$

$0.0416666666\ 66666666667\ z^4 + 0.0083333333\ 333333333334\ z^5 +$

$0.0013888888\ 888888888889\ z^6 + 0.0001984126\ 984126984127\ z^7 +$

$0.0000248015\ 87301587301587\ z^8 + 0.0000027557\ 319223985890653\ z^9 +$

$0.2755731922\ 3985890653E - 6\ z^{10} + O\left(z^{11}\right)$

<div align="center">

Type: UnivariateTaylorSeries(Float,z,0.0)

</div>

You can also create a power series by giving an explicit formula for its n-th coefficient. For details and more examples, see 8.9.6 on page 394.

To create a series about $w = 0$ whose n-th Taylor coefficient is $1/n!$, you can evaluate this expression. This is the Taylor expansion of $exp(w)$ at $w = 0$.

```
series(1/factorial(n),n,w = 0)
```

$$1 + w + \frac{1}{2}\ w^2 + \frac{1}{6}\ w^3 + \frac{1}{24}\ w^4 + \frac{1}{120}\ w^5 + \frac{1}{720}\ w^6 + \frac{1}{5040}\ w^7 +$$

$$\frac{1}{40320}\ w^8 + \frac{1}{362880}\ w^9 + \frac{1}{3628800}\ w^{10} + O\left(w^{11}\right)$$

<div align="center">

Type: UnivariatePuiseuxSeries(Expression Integer,w,0)

</div>

8.9.2   Coefficients of Power Series

You can extract any coefficient from a power series—even one that hasn't been computed yet. This is possible because in Axiom, infinite series are represented by a list of the coefficients that have already been determined, together with a function for computing the additional coefficients. (This is known as *lazy evaluation*.) When you ask for a coefficient that hasn't yet been computed, Axiom computes whatever additional coefficients it needs and then stores them in the representation of the power series.

Here's an example of how to extract the coefficients of a power series.

```
x := series(x)
```

$$x$$

```
y := exp(x) * sin(x)
```

$$x + x^2 + \tfrac{1}{3}\,x^3 - \tfrac{1}{30}\,x^5 - \tfrac{1}{90}\,x^6 - \tfrac{1}{630}\,x^7 + \tfrac{1}{22680}\,x^9 +$$

$$\frac{1}{113400}\,x^{10} + \frac{1}{1247400}\,x^{11} + O\left(x^{12}\right)$$

                  Type: UnivariatePuiseuxSeries(Expression Integer,x,0)

This coefficient is readily available.

```
coefficient(y,6)
```

$$-\frac{1}{90}$$

                                      Type: Expression Integer

But let's get the fifteenth coefficient of y.

```
coefficient(y,15)
```

$$-\frac{1}{10216206000}$$

                                      Type: Expression Integer

If you look at y then you see that the coefficients up to order 15 have all been
computed.

```
y
```

$$x + x^2 + \tfrac{1}{3}\,x^3 - \tfrac{1}{30}\,x^5 - \tfrac{1}{90}\,x^6 - \tfrac{1}{630}\,x^7 + \tfrac{1}{22680}\,x^9 + \tfrac{1}{113400}\,x^{10} +$$

$$\frac{1}{1247400}\,x^{11} - \frac{1}{97297200}\,x^{13} - \frac{1}{681080400}\,x^{14} - \frac{1}{10216206000}\,x^{15} + O\left(x^{16}\right)$$

                  Type: UnivariatePuiseuxSeries(Expression Integer,x,0)

8.9.3 Power Series Arithmetic

You can manipulate power series using the usual arithmetic operations $+$, $-$, $*$, and $/$ (from UnivariatePuiseuxSeries)

The results of these operations are also power series.

```
x := series x
```

$$x$$

> Type: UnivariatePuiseuxSeries(Expression Integer,x,0)

```
(3 + x) / (1 + 7*x)
```

$3 - 20\ x + 140\ x^2 - 980\ x^3 + 6860\ x^4 - 48020\ x^5 + 336140\ x^6 - 2352980\ x^7 +$

$16470860\ x^8 - 115296020\ x^9 + 807072140\ x^{10} + O\left(x^{11}\right)$

> Type: UnivariatePuiseuxSeries(Expression Integer,x,0)

You can also compute $f(x) * * g(x)$, where $f(x)$ and $g(x)$ are two power series.

```
base := 1 / (1 - x)
```

$$1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9 + x^{10} + O\left(x^{11}\right)$$

> Type: UnivariatePuiseuxSeries(Expression Integer,x,0)

```
expon := x * base
```

$$x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9 + x^{10} + x^{11} + O\left(x^{12}\right)$$

> Type: UnivariatePuiseuxSeries(Expression Integer,x,0)

```
base ** expon
```

$$1 + x^2 + \tfrac{3}{2}\ x^3 + \tfrac{7}{3}\ x^4 + \tfrac{43}{12}\ x^5 + \tfrac{649}{120}\ x^6 + \tfrac{241}{30}\ x^7 + \tfrac{3706}{315}\ x^8 +$$

$$\frac{85763}{5040}\ x^9 + \frac{245339}{10080}\ x^{10} + O\left(x^{11}\right)$$

> Type: UnivariatePuiseuxSeries(Expression Integer,x,0)

8.9.4   Functions on Power Series

Once you have created a power series, you can apply transcendental functions (for example, **exp**, **log**, **sin**, **tan**, **cosh**, etc.) to it.

To demonstrate this, we first create the power series expansion of the rational function

$$\frac{x^2}{1 - 6x + x^2}$$

about $x = 0$.

```
x := series 'x
```

$$x$$

Type: UnivariatePuiseuxSeries(Expression Integer,x,0)

```
rat := x**2 / (1 - 6*x + x**2)
```

$$x^2 + 6\ x^3 + 35\ x^4 + 204\ x^5 + 1189\ x^6 + 6930\ x^7 + 40391\ x^8 + 235416\ x^9 +$$

$$1372105\ x^{10} + 7997214\ x^{11} + 46611179\ x^{12} + O\left(x^{13}\right)$$

Type: UnivariatePuiseuxSeries(Expression Integer,x,0)

If you want to compute the series expansion of

$$\sin\left(\frac{x^2}{1 - 6x + x^2}\right)$$

you simply compute the sine of *rat*.

```
sin(rat)
```

$$x^2 + 6\ x^3 + 35\ x^4 + 204\ x^5 + \tfrac{7133}{6}\ x^6 + 6927\ x^7 + \tfrac{80711}{2}\ x^8 + 235068\ x^9 +$$

$$\frac{164285281}{120}\ x^{10} + \frac{31888513}{4}\ x^{11} + \frac{371324777}{8}\ x^{12} + O\left(x^{13}\right)$$

Type: UnivariatePuiseuxSeries(Expression Integer,x,0)

> **Warning:** the type of the coefficients of a power series may affect the kind of computations that you can do with that series. This can only happen when you have made a declaration to specify a series domain with a certain type of coefficient.

If you evaluate then you have declared that y is a one variable Taylor series (UTS is the abbreviation for `UnivariateTaylorSeries`) in the variable y with `FRAC INT` (that is, fractions of integer) coefficients, centered about 0.

```
y :  UTS(FRAC INT,y,0) := y
```

$$y$$

> Type: UnivariateTaylorSeries(Fraction Integer,y,0)

You can now compute certain power series in y, *provided* that these series have rational coefficients.

```
exp(y)
```

$$1 + y + \tfrac{1}{2}\, y^2 + \tfrac{1}{6}\, y^3 + \tfrac{1}{24}\, y^4 + \tfrac{1}{120}\, y^5 + \tfrac{1}{720}\, y^6 + \tfrac{1}{5040}\, y^7 + \tfrac{1}{40320}\, y^8 +$$

$$\frac{1}{362880}\, y^9 + \frac{1}{3628800}\, y^{10} + O\left(y^{11}\right)$$

> Type: UnivariateTaylorSeries(Fraction Integer,y,0)

You can get examples of such series by applying transcendental functions to series in y that have no constant terms.

```
tan(y**2)
```

$$y^2 + \frac{1}{3}\, y^6 + \frac{2}{15}\, y^{10} + O\left(y^{11}\right)$$

> Type: UnivariateTaylorSeries(Fraction Integer,y,0)

```
cos(y + y**5)
```

$$1 - \frac{1}{2}\, y^2 + \frac{1}{24}\, y^4 - \frac{721}{720}\, y^6 + \frac{6721}{40320}\, y^8 - \frac{1844641}{3628800}\, y^{10} + O\left(y^{11}\right)$$

> Type: UnivariateTaylorSeries(Fraction Integer,y,0)

Similarly, you can compute the logarithm of a power series with rational coefficients if the constant coefficient is 1.

```
log(1 + sin(y))
```

$$y - \frac{1}{2}\, y^2 + \frac{1}{6}\, y^3 - \frac{1}{12}\, y^4 + \frac{1}{24}\, y^5 - \frac{1}{45}\, y^6 + \frac{61}{5040}\, y^7 - \frac{17}{2520}\, y^8 + \frac{277}{72576}\, y^9 -$$

$$\frac{31}{14175}\, y^{10} + O\left(y^{11}\right)$$

<div align="right">Type: UnivariateTaylorSeries(Fraction Integer,y,0)</div>

If you wanted to apply, say, the operation **exp** to a power series with a nonzero constant coefficient a_0, then the constant coefficient of the result would be e^{a_0}, which is *not* a rational number. Therefore, evaluating $exp(2 + tan(y))$ would generate an error message.

If you want to compute the Taylor expansion of $exp(2+tan(y))$, you must ensure that the coefficient domain has an operation **exp** defined for it. An example of such a domain is `Expression Integer`, the type of formal functional expressions over the integers.

When working with coefficients of this type,

```
z :  UTS(EXPR INT,z,0) := z
```

$$z$$

<div align="right">Type: UnivariateTaylorSeries(Expression Integer,z,0)</div>

this presents no problems.

```
exp(2 + tan(z))
```

$$e^2 + e^2\, z + \frac{e^2}{2}\, z^2 + \frac{e^2}{2}\, z^3 + \frac{3\, e^2}{8}\, z^4 + \frac{37\, e^2}{120}\, z^5 + \frac{59\, e^2}{240}\, z^6 + \frac{137\, e^2}{720}\, z^7 +$$

$$\frac{871\, e^2}{5760}\, z^8 + \frac{41641\, e^2}{362880}\, z^9 + \frac{325249\, e^2}{3628800}\, z^{10} + O\left(z^{11}\right)$$

<div align="right">Type: UnivariateTaylorSeries(Expression Integer,z,0)</div>

Another way to create Taylor series whose coefficients are expressions over the integers is to use **taylor** which works similarly to **series**.

This is equivalent to the previous computation, except that now we are using the variable w instead of z.

```
w := taylor 'w
```

$$w$$

                Type: UnivariateTaylorSeries(Expression Integer,w,0)

```
exp(2 + tan(w))
```

$$e^2 + e^2\ w + \frac{e^2}{2}\ w^2 + \frac{e^2}{2}\ w^3 + \frac{3\ e^2}{8}\ w^4 + \frac{37\ e^2}{120}\ w^5 + \frac{59\ e^2}{240}\ w^6 + \frac{137\ e^2}{720}\ w^7 +$$

$$\frac{871\ e^2}{5760}\ w^8 + \frac{41641\ e^2}{362880}\ w^9 + \frac{325249\ e^2}{3628800}\ w^{10} + O\left(w^{11}\right)$$

                Type: UnivariateTaylorSeries(Expression Integer,w,0)

8.9.5 Converting to Power Series

The `ExpressionToUnivariatePowerSeries` package provides operations for computing series expansions of functions.

Evaluate this to compute the Taylor expansion of $sin x$ about $x = 0$. The first argument, $sin(x)$, specifies the function whose series expansion is to be computed and the second argument, $x = 0$, specifies that the series is to be expanded in power of $(x - 0)$, that is, in power of x.

```
taylor(sin(x),x = 0)
```

$$x - \frac{1}{6}\ x^3 + \frac{1}{120}\ x^5 - \frac{1}{5040}\ x^7 + \frac{1}{362880}\ x^9 + O\left(x^{11}\right)$$

                Type: UnivariateTaylorSeries(Expression Integer,x,0)

Here is the Taylor expansion of $sin x$ about $x = \frac{\pi}{6}$:

```
taylor(sin(x),x = %pi/6)
```

$$\frac{1}{2} + \frac{\sqrt{3}}{2}\left(x - \frac{\pi}{6}\right) - \frac{1}{4}\left(x - \frac{\pi}{6}\right)^2 - \frac{\sqrt{3}}{12}\left(x - \frac{\pi}{6}\right)^3 + \frac{1}{48}\left(x - \frac{\pi}{6}\right)^4 +$$

$$\frac{\sqrt{3}}{240}\left(x - \frac{\pi}{6}\right)^5 - \frac{1}{1440}\left(x - \frac{\pi}{6}\right)^6 - \frac{\sqrt{3}}{10080}\left(x - \frac{\pi}{6}\right)^7 + \frac{1}{80640}\left(x - \frac{\pi}{6}\right)^8 +$$

$$\frac{\sqrt{3}}{725760}\left(x - \frac{\pi}{6}\right)^9 - \frac{1}{7257600}\left(x - \frac{\pi}{6}\right)^{10} + O\left(\left(x - \frac{\pi}{6}\right)^{11}\right)$$

```
      Type:   UnivariateTaylorSeries(Expression Integer,x,pi/6)
```

The function to be expanded into a series may have variables other than the series variable.

For example, we may expand $tan(x*y)$ as a Taylor series in x

```
taylor(tan(x*y),x = 0)
```

$$y\,x + \frac{y^3}{3}\,x^3 + \frac{2\,y^5}{15}\,x^5 + \frac{17\,y^7}{315}\,x^7 + \frac{62\,y^9}{2835}\,x^9 + O\left(x^{11}\right)$$

```
         Type:   UnivariateTaylorSeries(Expression Integer,x,0)
```

or as a Taylor series in y.

```
taylor(tan(x*y),y = 0)
```

$$x\,y + \frac{x^3}{3}\,y^3 + \frac{2\,x^5}{15}\,y^5 + \frac{17\,x^7}{315}\,y^7 + \frac{62\,x^9}{2835}\,y^9 + O\left(y^{11}\right)$$

```
         Type:   UnivariateTaylorSeries(Expression Integer,y,0)
```

A more interesting function is
$$\frac{te^{xt}}{e^t - 1}$$

When we expand this function as a Taylor series in t the n-th order coefficient is the n-th Bernoulli polynomial divided by $n!$.

```
bern := taylor(t*exp(x*t)/(exp(t) - 1),t = 0)
```

$$1 + \frac{2\,x - 1}{2}\,t + \frac{6\,x^2 - 6\,x + 1}{12}\,t^2 + \frac{2\,x^3 - 3\,x^2 + x}{12}\,t^3 +$$

$$\frac{30\,x^4 - 60\,x^3 + 30\,x^2 - 1}{720}\,t^4 + \frac{6\,x^5 - 15\,x^4 + 10\,x^3 - x}{720}\,t^5 +$$

$$\frac{42\,x^6 - 126\,x^5 + 105\,x^4 - 21\,x^2 + 1}{30240}\,t^6 + \frac{6\,x^7 - 21\,x^6 + 21\,x^5 - 7\,x^3 + x}{30240}\,t^7 +$$

$$\frac{30\,x^8 - 120\,x^7 + 140\,x^6 - 70\,x^4 + 20\,x^2 - 1}{1209600}\,t^8 +$$

$$\frac{10\,x^9 - 45\,x^8 + 60\,x^7 - 42\,x^5 + 20\,x^3 - 3\,x}{3628800}\,t^9 +$$

$$\frac{66\,x^{10} - 330\,x^9 + 495\,x^8 - 462\,x^6 + 330\,x^4 - 99\,x^2 + 5}{239500800}\,t^{10} + O\left(t^{11}\right)$$

$$\text{Type: } \texttt{UnivariateTaylorSeries(Expression Integer,t,0)}$$

Therefore, this and the next expression produce the same result.

```
factorial(6) * coefficient(bern,6)
```

$$\frac{42\ x^6 - 126\ x^5 + 105\ x^4 - 21\ x^2 + 1}{42}$$

$$\text{Type: } \text{ Expression Integer}$$

```
bernoulliB(6,x)
```

$$x^6 - 3\ x^5 + \frac{5}{2}\ x^4 - \frac{1}{2}\ x^2 + \frac{1}{42}$$

$$\text{Type: } \text{ Polynomial Fraction Integer}$$

Technically, a series with terms of negative degree is not considered to be a Taylor series, but, rather, a *Laurent series*. If you try to compute a Taylor series expansion of $\frac{x}{\log x}$ at $x = 1$ via $taylor(x/log(x), x = 1)$ you get an error message. The reason is that the function has a *pole* at $x = 1$, meaning that its series expansion about this point has terms of negative degree. A series with finitely many terms of negative degree is called a Laurent series.

You get the desired series expansion by issuing this.

```
laurent(x/log(x),x = 1)
```

$$(x - 1)^{(-1)} + \frac{3}{2} + \frac{5}{12}\ (x - 1) - \frac{1}{24}\ (x - 1)^2 + \frac{11}{720}\ (x - 1)^3 - \frac{11}{1440}\ (x - 1)^4 +$$

$$\frac{271}{60480}\ (x - 1)^5 - \frac{13}{4480}\ (x - 1)^6 + \frac{7297}{3628800}\ (x - 1)^7 - \frac{425}{290304}\ (x - 1)^8 +$$

$$\frac{530113}{479001600}\ (x - 1)^9 + O\left((x - 1)^{10}\right)$$

$$\text{Type: } \texttt{UnivariateLaurentSeries(Expression Integer,x,1)}$$

Similarly, a series with terms of fractional degree is neither a Taylor series nor a Laurent series. Such a series is called a *Puiseux series*. The expression $laurent(sqrt(sec(x)), x = 3 * \%pi/2)$ results in an error message because the series expansion about this point has terms of fractional degree.

However, this command produces what you want.

```
puiseux(sqrt(sec(x)),x = 3 * %pi/2)
```

$$\left(x - \frac{3\,\pi}{2}\right)^{\left(-\frac{1}{2}\right)} + \frac{1}{12}\,\left(x - \frac{3\,\pi}{2}\right)^{\frac{3}{2}} + \frac{1}{160}\,\left(x - \frac{3\,\pi}{2}\right)^{\frac{7}{2}} + O\left(\left(x - \frac{3\,\pi}{2}\right)^{5}\right)$$

Type: UnivariatePuiseuxSeries(Expression Integer,x,(3*pi)/2)

Finally, consider the case of functions that do not have Puiseux expansions about certain points. An example of this is x^x about $x = 0$. $puiseux(x**x, x = 0)$ produces an error message because of the type of singularity of the function at $x = 0$.

The general function **series** can be used in this case. Notice that the series returned is not, strictly speaking, a power series because of the $log(x)$ in the expansion.

```
series(x**x,x=0)
```

$$1 + \log\left(x\right)\,x + \frac{\log(x)^2}{2}\,x^2 + \frac{\log(x)^3}{6}\,x^3 + \frac{\log(x)^4}{24}\,x^4 + \frac{\log(x)^5}{120}\,x^5 + \frac{\log(x)^6}{720}\,x^6 +$$

$$\frac{\log\left(x\right)^7}{5040}\,x^7 + \frac{\log\left(x\right)^8}{40320}\,x^8 + \frac{\log\left(x\right)^9}{362880}\,x^9 + \frac{\log\left(x\right)^{10}}{3628800}\,x^{10} + O\left(x^{11}\right)$$

Type: GeneralUnivariatePowerSeries(Expression Integer,x,0)

> The operation **series** returns the most general type of infinite series. The user who is not interested in distinguishing between various types of infinite series may wish to use this operation exclusively.

8.9.6   Power Series from Formulas

The `GenerateUnivariatePowerSeries` package enables you to create power series from explicit formulas for their n-th coefficients. In what follows, we construct series expansions for certain transcendental functions by giving formulas for their coefficients. You can also compute such series expansions directly simply by specifying the function and the point about which the series is to be expanded. See 8.9.5 on page 391 for more information.

Consider the Taylor expansion of e^x about $x = 0$:

$$\begin{aligned} e^x &= 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \cdots \\ &= \sum_{n=0}^{\infty} \frac{x^n}{n!} \end{aligned}$$

The n-th Taylor coefficient is $1/n!$.

This is how you create this series in Axiom.

```
series(n +-> 1/factorial(n),x = 0)
```

$$1 + x + \tfrac{1}{2}\,x^2 + \tfrac{1}{6}\,x^3 + \tfrac{1}{24}\,x^4 + \tfrac{1}{120}\,x^5 + \tfrac{1}{720}\,x^6 + \tfrac{1}{5040}\,x^7 + \tfrac{1}{40320}\,x^8 +$$

$$\frac{1}{362880}\,x^9 + \frac{1}{3628800}\,x^{10} + O\left(x^{11}\right)$$

Type: UnivariatePuiseuxSeries(Expression Integer,x,0)

The first argument specifies a formula for the n-th coefficient by giving a function that maps n to $1/n!$. The second argument specifies that the series is to be expanded in powers of $(x - 0)$, that is, in powers of x. Since we did not specify an initial degree, the first term in the series was the term of degree 0 (the constant term). Note that the formula was given as an anonymous function. These are discussed in 6.17 on page 275.

Consider the Taylor expansion of $logx$ about $x = 1$:

$$\log(x) \;=\; (x - 1) - \frac{(x - 1)^2}{2} + \frac{(x - 1)^3}{3} - \cdots$$

$$=\; \sum_{n=1}^{\infty}(-1)^{n-1}\frac{(x - 1)^n}{n}$$

If you were to evaluate the expression $series(n + -> (-1)**(n-1)/n, x = 1)$ you would get an error message because Axiom would try to calculate a term of degree 0 and therefore divide by 0.

Instead, evaluate this. The third argument, 1.., indicates that only terms of degree $n = 1, \ldots$ are to be computed.

```
series(n +-> (-1)**(n-1)/n,x = 1,1..)
```

$$(x - 1) - \tfrac{1}{2}\,(x - 1)^2 + \tfrac{1}{3}\,(x - 1)^3 - \tfrac{1}{4}\,(x - 1)^4 + \tfrac{1}{5}\,(x - 1)^5 - \tfrac{1}{6}\,(x - 1)^6 +$$

$$\frac{1}{7}\,(x - 1)^7 - \frac{1}{8}\,(x - 1)^8 + \frac{1}{9}\,(x - 1)^9 - \frac{1}{10}\,(x - 1)^{10} + \frac{1}{11}\,(x - 1)^{11} +$$

$$O\left((x - 1)^{12}\right)$$

Type: UnivariatePuiseuxSeries(Expression Integer,x,1)

Next consider the Taylor expansion of an odd function, say, $sin(x)$:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots$$

Here every other coefficient is zero and we would like to give an explicit formula only for the odd Taylor coefficients.

This is one way to do it. The third argument, 1.., specifies that the first term to be computed is the term of degree 1. The fourth argument, 2, specifies that we increment by 2 to find the degrees of subsequent terms, that is, the next term is of degree $1 + 2$, the next of degree $1 + 2 + 2$, etc.

```
series(n +-> (-1)**((n-1)/2)/factorial(n),x = 0,1..,2)
```

$$x - \frac{1}{6}\,x^3 + \frac{1}{120}\,x^5 - \frac{1}{5040}\,x^7 + \frac{1}{362880}\,x^9 - \frac{1}{39916800}\,x^{11} + O\left(x^{12}\right)$$

<div align="center">Type: UnivariatePuiseuxSeries(Expression Integer,x,0)</div>

The initial degree and the increment do not have to be integers. For example, this expression produces a series expansion of $\sin(x^{\frac{1}{3}})$.

```
series(n +-> (-1)**((3*n-1)/2)/factorial(3*n),x = 0,1/3..,2/3)
```

$$x^{\frac{1}{3}} - \frac{1}{6}\,x + \frac{1}{120}\,x^{\frac{5}{3}} - \frac{1}{5040}\,x^{\frac{7}{3}} + \frac{1}{362880}\,x^3 - \frac{1}{39916800}\,x^{\frac{11}{3}} + O\left(x^4\right)$$

<div align="center">Type: UnivariatePuiseuxSeries(Expression Integer,x,0)</div>

While the increment must be positive, the initial degree may be negative. This yields the Laurent expansion of $csc(x)$ at $x = 0$. (bernoulli(numer(n+1)) is necessary because bernoulli takes integer arguments.)

```
cscx := series(n +-> (-1)**((n-1)/2) * 2 * (2**n-1) *
bernoulli(numer(n+1)) / factorial(n+1), x=0, -1..,2)
```

$$x^{(-1)} + \frac{1}{6}\,x + \frac{7}{360}\,x^3 + \frac{31}{15120}\,x^5 + \frac{127}{604800}\,x^7 + \frac{73}{3421440}\,x^9 + O\left(x^{10}\right)$$

<div align="center">Type: UnivariatePuiseuxSeries(Expression Integer,x,0)</div>

Of course, the reciprocal of this power series is the Taylor expansion of $sin(x)$.

```
1/cscx
```

$$x - \frac{1}{6}\, x^3 + \frac{1}{120}\, x^5 - \frac{1}{5040}\, x^7 + \frac{1}{362880}\, x^9 - \frac{1}{39916800}\, x^{11} + O\left(x^{12}\right)$$

```
Type:  UnivariatePuiseuxSeries(Expression Integer,x,0)
```

As a final example,here is the Taylor expansion of $asin(x)$ about $x = 0$.

```
asinx := series(n +->
binomial(n-1,(n-1)/2)/(n*2**(n-1)),x=0,1..,2)
```

$$x + \frac{1}{6}\, x^3 + \frac{3}{40}\, x^5 + \frac{5}{112}\, x^7 + \frac{35}{1152}\, x^9 + \frac{63}{2816}\, x^{11} + O\left(x^{12}\right)$$

```
Type:  UnivariatePuiseuxSeries(Expression Integer,x,0)
```

When we compute the *sin* of this series, we get x (in the sense that all higher terms computed so far are zero).

```
sin(asinx)
```

$$x + O\left(x^{12}\right)$$

```
Type:  UnivariatePuiseuxSeries(Expression Integer,x,0)
```

Axiom isn't sufficiently "symbolic" in the sense we might wish. It is an open problem to decide that "x" is the only surviving term. Two attacks on the problem might be:

(1) Notice that all of the higher terms are identically zero but Axiom can't decide that from the information it knows. Presumably we could attack this problem by looking at the sin function as a taylor series around x=0 and seeing the term cancellation occur. This uses a term-difference mechanism.

(2) Notice that there is no way to decide that the stream for asinx is actually the definition of asin(x). But we could recognize that the stream for asin(x) has a generator term and so will a taylor series expansion of sin(x). From these two generators it may be possible in certain cases to decide that the application of one generator to the other will yield only "x". This trick involves finding the correct inverse for the stream functions. If we can find an inverse for the "remaining tail" of the stream we could conclude cancellation and thus turn an infinite stream into a finite object.

In general this is the zero-equivalence problem and is undecidable.

As we discussed in 8.9.5 on page 391, you can also use the operations **taylor**, **laurent** and **puiseux** instead of **series** if you know ahead of time what kind of exponents a series has. You can't go wrong using **series**, though.

8.9.7   Substituting Numerical Values in Power Series

Use **eval** to substitute a numerical value for a variable in a power series. For example, here's a way to obtain numerical approximations of $\%e$ from the Taylor series expansion of $exp(x)$.

First you create the desired Taylor expansion.

```
f := taylor(exp(x))
```

$$1 + x + \frac{1}{2}\ x^2 + \frac{1}{6}\ x^3 + \frac{1}{24}\ x^4 + \frac{1}{120}\ x^5 + \frac{1}{720}\ x^6 + \frac{1}{5040}\ x^7 +$$

$$\frac{1}{40320}\ x^8 + \frac{1}{362880}\ x^9 + \frac{1}{3628800}\ x^{10} + O\left(x^{11}\right)$$

<div align="center">Type: UnivariateTaylorSeries(Expression Integer,x,0)</div>

Then you evaluate the series at the value 1.0. The result is a sequence of the partial sums.

```
eval(f,1.0)
```

$$[1.0, 2.0, 2.5, 2.6666666666\ 666666667, 2.7083333333\ 333333333,$$

$$2.7166666666\ 666666667, 2.7180555555\ 555555556, 2.7182539682\ 53968254,$$

$$2.7182787698\ 412698413, 2.7182815255\ 731922399, \ldots]$$

<div align="center">Type: Stream Expression Float</div>

8.9.8   Example: Bernoulli Polynomials and Sums of Powers

Axiom provides operations for computing definite and indefinite sums.

You can compute the sum of the first ten fourth powers by evaluating this. This creates a list whose entries are m^4 as m ranges from 1 to 10, and then computes the sum of the entries of that list.

```
reduce(+,[m**4 for m in 1..10])
```

<div align="center">25333</div>

<div align="center">Type: PositiveInteger</div>

You can also compute a formula for the sum of the first k fourth powers, where k is an unspecified positive integer.

```
sum4 := sum(m**4, m = 1..k)
```

$$\frac{6\ k^5 + 15\ k^4 + 10\ k^3 - k}{30}$$

Type: Fraction Polynomial Integer

This formula is valid for any positive integer k. For instance, if we replace k by 10, we obtain the number we computed earlier.

```
eval(sum4, k = 10)
```

$$25333$$

Type: Fraction Polynomial Integer

You can compute a formula for the sum of the first k n-th powers in a similar fashion. Just replace the 4 in the definition of **sum4** by any expression not involving k. Axiom computes these formulas using Bernoulli polynomials; we use the rest of this section to describe this method.

First consider this function of t and x.

```
f := t*exp(x*t) / (exp(t) - 1)
```

$$\frac{t\ e^{(t\ x)}}{e^t - 1}$$

Type: Expression Integer

Since the expressions involved get quite large, we tell Axiom to show us only terms of degree up to 5.

```
)set streams calculate 5
```

If we look at the Taylor expansion of $f(x, t)$ about $t = 0$, we see that the coefficients of the powers of t are polynomials in x.

```
ff := taylor(f,t = 0)
```

$$1 + \frac{2\ x - 1}{2}\ t + \frac{6\ x^2 - 6\ x + 1}{12}\ t^2 + \frac{2\ x^3 - 3\ x^2 + x}{12}\ t^3 +$$

$$\frac{30\ x^4 - 60\ x^3 + 30\ x^2 - 1}{720}\ t^4 + \frac{6\ x^5 - 15\ x^4 + 10\ x^3 - x}{720}\ t^5 + O\left(t^6\right)$$

Type: UnivariateTaylorSeries(Expression Integer,t,0)

In fact, the n-th coefficient in this series is essentially the n-th Bernoulli polynomial: the n-th coefficient of the series is $\frac{1}{n!}B_n(x)$, where $B_n(x)$ is the n-th Bernoulli polynomial. Thus, to obtain the n-th Bernoulli polynomial, we multiply the n-th coefficient of the series ff by $n!$.

For example, the sixth Bernoulli polynomial is this.

```
factorial(6) * coefficient(ff,6)
```

$$\frac{42\ x^6 - 126\ x^5 + 105\ x^4 - 21\ x^2 + 1}{42}$$

Type: Expression Integer

We derive some properties of the function $f(x, t)$. First we compute $f(x+1, t) - f(x, t)$.

```
g := eval(f, x = x + 1) - f
```

$$\frac{t\ e^{(t\ x + t)} - t\ e^{(t\ x)}}{e^t - 1}$$

Type: Expression Integer

If we normalize g, we see that it has a particularly simple form.

```
normalize(g)
```

$$t\ e^{(t\ x)}$$

Type: Expression Integer

From this it follows that the n-th coefficient in the Taylor expansion of $g(x, t)$ at $t = 0$ is

$$\frac{1}{(n-1)!}x^{n-1}$$

.

If you want to check this, evaluate the next expression.

```
taylor(g,t = 0)
```

$$t + x \, t^2 + \frac{x^2}{2} \, t^3 + \frac{x^3}{6} \, t^4 + \frac{x^4}{24} \, t^5 + O\left(t^6\right)$$

```
Type:   UnivariateTaylorSeries(Expression Integer,t,0)
```

However, since

$$g(x,t) = f(x+1,t) - f(x,t)$$

it follows that the n-th coefficient is

$$\frac{1}{n!}(B_n(x+1) - B_n(x))$$

Equating coefficients, we see that

$$\frac{1}{(n-1)!}x^{n-1} = \frac{1}{n!}(B_n(x+1) - B_n(x))$$

and, therefore,

$$x^{n-1} = \frac{1}{n}(B_n(x+1) - B_n(x))$$

Let's apply this formula repeatedly, letting x vary between two integers a and b, with $a < b$:

$$
\begin{aligned}
a^{n-1} &= \tfrac{1}{n}(B_n(a+1) - B_n(a)) \\
(a+1)^{n-1} &= \tfrac{1}{n}(B_n(a+2) - B_n(a+1)) \\
(a+2)^{n-1} &= \tfrac{1}{n}(B_n(a+3) - B_n(a+2)) \\
&\vdots \\
(b-1)^{n-1} &= \tfrac{1}{n}(B_n(b) - B_n(b-1)) \\
b^{n-1} &= \tfrac{1}{n}(B_n(b+1) - B_n(b))
\end{aligned}
$$

When we add these equations we find that the sum of the left-hand sides is

$$\sum_{m=a}^{b} m^{n-1},$$

the sum of the

$$(n-1)^{\text{st}}$$

powers from a to b. The sum of the right-hand sides is a "telescoping series." After cancellation, the sum is simply

$$\frac{1}{n}(B_n(b+1) - B_n(a))$$

Replacing n by $n + 1$, we have shown that

$$\sum_{m=a}^{b} m^n = \frac{1}{n+1}(B_{n+1}(b+1) - B_{n+1}(a))$$

Let's use this to obtain the formula for the sum of fourth powers.

First we obtain the Bernoulli polynomial B_5.

```
B5 := factorial(5) * coefficient(ff,5)
```

$$\frac{6\ x^5 - 15\ x^4 + 10\ x^3 - x}{6}$$

<div align="right">Type: Expression Integer</div>

To find the sum of the first k 4th powers, we multiply $1/5$ by $B_5(k+1) - B_5(1)$.

```
1/5 * (eval(B5, x = k + 1) - eval(B5, x = 1))
```

$$\frac{6\ k^5 + 15\ k^4 + 10\ k^3 - k}{30}$$

<div align="right">Type: Expression Integer</div>

This is the same formula that we obtained via $sum(m**4, m = 1..k)$.

```
sum4
```

$$\frac{6\ k^5 + 15\ k^4 + 10\ k^3 - k}{30}$$

<div align="right">Type: Fraction Polynomial Integer</div>

At this point you may want to do the same computation, but with an exponent other than 4. For example, you might try to find a formula for the sum of the first k 20th powers.

8.10   Solution of Differential Equations

In this section we discuss Axiom's facilities for solving differential equations in closed-form and in series.

Axiom provides facilities for closed-form solution of single differential equations of the following kinds:

- linear ordinary differential equations, and

- non-linear first order ordinary differential equations when integrating factors can be found just by integration.

For a discussion of the solution of systems of linear and polynomial equations, see 8.5 on page 366.

8.10.1 Closed-Form Solutions of Linear Differential Equations

A *differential equation* is an equation involving an unknown *function* and one or more of its derivatives. The equation is called *ordinary* if derivatives with respect to only one dependent variable appear in the equation (it is called *partial* otherwise). The package `ElementaryFunctionODESolver` provides the top-level operation **solve** for finding closed-form solutions of ordinary differential equations.

To solve a differential equation, you must first create an operator for the unknown function.

We let y be the unknown function in terms of x.

```
y := operator 'y
```

$$y$$

<div align="right">Type: BasicOperator</div>

You then type the equation using D to create the derivatives of the unknown function $y(x)$ where x is any symbol you choose (the so-called *dependent variable*).

This is how you enter the equation $y'' + y' + y = 0$.

```
deq := D(y x, x, 2) + D(y x, x) + y x = 0
```

$$y''(x) + y'(x) + y(x) = 0$$

<div align="right">Type: Equation Expression Integer</div>

The simplest way to invoke the **solve** command is with three arguments.

- the differential equation,

- the operator representing the unknown function,

- the dependent variable.

So, to solve the above equation, we enter this.

```
solve(deq, y, x)
```

$$\left[particular = 0, basis = \left[\cos\left(\frac{x\sqrt{3}}{2}\right) e^{\left(-\frac{x}{2}\right)}, e^{\left(-\frac{x}{2}\right)} \sin\left(\frac{x\sqrt{3}}{2}\right)\right]\right]$$

    Type: Union(Record(particular: Expression Integer,basis: List
                                       Expression Integer),...)

Since linear ordinary differential equations have infinitely many solutions, **solve**
returns a *particular solution* f_p and a basis $f_1, \ldots, f_n$ for the solutions of the
corresponding homogenuous equation. Any expression of the form

$$f_p + c_1 f_1 + \ldots c_n f_n$$

where the c_i do not involve the dependent variable is also a solution. This is
similar to what you get when you solve systems of linear algebraic equations.

A way to select a unique solution is to specify *initial conditions*: choose a value
a for the dependent variable and specify the values of the unknown function and
its derivatives at a. If the number of initial conditions is equal to the order of
the equation, then the solution is unique (if it exists in closed form!) and **solve**
tries to find it. To specify initial conditions to **solve**, use an `Equation` of the
form $x = a$ for the third parameter instead of the dependent variable, and add
a fourth parameter consisting of the list of values $y(a), y'(a), \ldots$.

To find the solution of $y'' + y = 0$ satisfying $y(0) = y'(0) = 1$, do this.

```
deq := D(y x, x, 2) + y x
```

$$y''(x) + y(x)$$

                                    Type: Expression Integer

You can omit the $= 0$ when you enter the equation to be solved.

```
solve(deq, y, x = 0, [1, 1])
```

$$\sin(x) + \cos(x)$$

                        Type: Union(Expression Integer,...)

Axiom is not limited to linear differential equations with constant coefficients. It can also find solutions when the coefficients are rational or algebraic functions of the dependent variable. Furthermore, Axiom is not limited by the order of the equation.

Axiom can solve the following third order equations with polynomial coefficients.

```
deq := x**3 * D(y x, x, 3) + x**2 * D(y x, x, 2) - 2 * x * D(y x,
x) + 2 * y x = 2 * x**4
```

$$x^3 \ y''' \ (x) + x^2 \ y'' \ (x) - 2 \ x \ y' \ (x) + 2 \ y \ (x) = 2 \ x^4$$

<div align="right">Type: Equation Expression Integer</div>

```
solve(deq, y, x)
```

$$\left[particular = \frac{x^5 - 10 \ x^3 + 20 \ x^2 + 4}{15 \ x}, \right.$$

$$\left. basis = \left[\frac{2 \ x^3 - 3 \ x^2 + 1}{x}, \frac{x^3 - 1}{x}, \frac{x^3 - 3 \ x^2 - 1}{x} \right] \right]$$

<div align="right">Type: Union(Record(particular: Expression Integer,basis: List
Expression Integer),...)</div>

Here we are solving a homogeneous equation.

```
deq := (x**9+x**3) * D(y x, x, 3) + 18 * x**8 * D(y x, x, 2) - 90
* x * D(y x, x) - 30 * (11 * x**6 - 3) * y x
```

$$\left(x^9 + x^3\right) \ y''' \ (x) + 18 \ x^8 \ y'' \ (x) - 90 \ x \ y' \ (x) + \left(-330 \ x^6 + 90\right) \ y \ (x)$$

<div align="right">Type: Expression Integer</div>

```
solve(deq, y, x)
```

$$\left[particular = 0, basis = \left[\frac{x}{x^6 + 1}, \frac{x \ e^{\left(-\sqrt{91} \ \log(x)\right)}}{x^6 + 1}, \frac{x \ e^{\left(\sqrt{91} \ \log(x)\right)}}{x^6 + 1} \right] \right]$$

<div align="right">Type: Union(Record(particular: Expression Integer,basis: List
Expression Integer),...)</div>

On the other hand, and in contrast with the operation **integrate**, it can happen that Axiom finds no solution and that some closed-form solution still exists. While it is mathematically complicated to describe exactly when the solutions are guaranteed to be found, the following statements are correct and form good guidelines for linear ordinary differential equations:

- If the coefficients are constants, Axiom finds a complete basis of solutions (i,e, all solutions).

- If the coefficients are rational functions in the dependent variable, Axiom at least finds all solutions that do not involve algebraic functions.

Note that this last statement does not mean that Axiom does not find the solutions that are algebraic functions. It means that it is not guaranteed that the algebraic function solutions will be found.

This is an example where all the algebraic solutions are found.

```
deq := (x**2 + 1) * D(y x, x, 2) + 3 * x * D(y x, x) + y x = 0
```

$$\left(x^2 + 1\right) \, y'' \, (x) + 3 \, x \, y' \, (x) + y \, (x) = 0$$

<div align="right">Type: Equation Expression Integer</div>

```
solve(deq, y, x)
```

$$\left[particular = 0, basis = \left[\frac{1}{\sqrt{x^2 + 1}}, \frac{\log\left(\sqrt{x^2 + 1} - x\right)}{\sqrt{x^2 + 1}}\right]\right]$$

<div align="right">Type: Union(Record(particular: Expression Integer,basis: List
Expression Integer),...)</div>

8.10.2   Closed-Form Solutions of Non-Linear Differential Equations

This is an example that shows how to solve a non-linear first order ordinary differential equation manually when an integrating factor can be found just by integration. At the end, we show you how to solve it directly.

Let's solve the differential equation $y' = y/(x + y \log y)$.

Using the notation $m(x, y) + n(x, y)y' = 0$, we have $m = -y$ and $n = x + y \log y$.

```
m := -y
```

$$-y$$

`n := x + y * log y`

$$y \log{(y)} + x$$

We first check for exactness, that is, does $dm/dy = dn/dx$?

`D(m, y) - D(n, x)`

$$-2$$

This is not zero, so the equation is not exact. Therefore we must look for an integrating factor: a function $mu(x, y)$ such that $d(mum)/dy = d(mun)/dx$. Normally, we first search for $mu(x, y)$ depending only on x or only on y.

Let's search for such a $mu(x)$ first.

`mu := operator 'mu`

$$mu$$

`a := D(mu(x) * m, y) - D(mu(x) * n, x)`

$$(-y \log{(y)} - x) \, mu'\,(x) - 2 \, mu\,(x)$$

If the above is zero for a function mu that does *not* depend on y, then $mu(x)$ is an integrating factor.

`solve(a = 0, mu, x)`

$$\left[particular = 0, basis = \left[\frac{1}{y^2 \, \log{(y)}^2 + 2 \, x \, y \, \log{(y)} + x^2} \right] \right]$$

```
  Type:  Union(Record(particular:  Expression Integer,basis:  List
                                         Expression Integer),...)
```

The solution depends on y, so there is no integrating factor that depends on x only.

Let's look for one that depends on y only.

```
b  := D(mu(y) * m, y) - D(mu(y) * n, x)
```

$$-y \; mu^{,} \, (y) - 2 \; mu \, (y)$$

```
                                   Type:  Expression Integer
```

```
sb := solve(b = 0, mu, y)
```

$$\left[particular = 0, basis = \left[\frac{1}{y^2} \right] \right]$$

```
  Type:  Union(Record(particular:  Expression Integer,basis:  List
                                         Expression Integer),...)
```

We've found one!

The above $mu(y)$ is an integrating factor. We must multiply our initial equation (that is, m and n) by the integrating factor.

```
intFactor := sb.basis.1
```

$$\frac{1}{y^2}$$

```
                                   Type:  Expression Integer
```

```
m := intFactor * m
```

$$-\frac{1}{y}$$

```
                                   Type:  Expression Integer
```

```
n := intFactor * n
```

$$\frac{y \log{(y)} + x}{y^2}$$

                                        Type: Expression Integer

Let's check for exactness.

```
D(m, y) - D(n, x)
```

$$0$$

                                        Type: Expression Integer

We must solve the exact equation, that is, find a function $s(x, y)$ such that $ds/dx = m$ and $ds/dy = n$.

We start by writing $s(x, y) = h(y) + integrate(m, x)$ where $h(y)$ is an unknown function of y. This guarantees that $ds/dx = m$.

```
h := operator 'h
```

$$h$$

                                        Type: BasicOperator

```
sol := h y + integrate(m, x)
```

$$\frac{y\, h(y) - x}{y}$$

                                        Type: Expression Integer

All we want is to find $h(y)$ such that $ds/dy = n$.

```
dsol := D(sol, y)
```

$$\frac{y^2\, h'(y) + x}{y^2}$$

                                        Type: Expression Integer

```
nsol := solve(dsol = n, h, y)
```

$$\left[particular = \frac{\log{(y)}^2}{2}, basis = [1]\right]$$

```
Type:  Union(Record(particular:  Expression Integer,basis:  List
                                        Expression Integer),...)
```

The above particular solution is the $h(y)$ we want, so we just replace $h(y)$ by it in the implicit solution.

```
eval(sol, h y = nsol.particular)
```

$$\frac{y \log{(y)}^2 - 2\ x}{2\ y}$$

```
Type:  Expression Integer
```

A first integral of the initial equation is obtained by setting this result equal to an arbitrary constant.

Now that we've seen how to solve the equation "by hand," we show you how to do it with the **solve** operation.

First define y to be an operator.

```
y := operator 'y
```

$$y$$

```
Type:  BasicOperator
```

Next we create the differential equation.

```
deq := D(y x, x) = y(x) / (x + y(x) * log y x)
```

$$y'\ (x) = \frac{y\ (x)}{y\ (x)\ \log{(y\ (x))} + x}$$

```
Type:  Equation Expression Integer
```

Finally, we solve it.

```
solve(deq, y, x)
```

$$\frac{y\ (x)\ \log{(y\ (x))}^2 - 2\ x}{2\ y\ (x)}$$

```
Type:  Union(Expression Integer,...)
```

8.10.3 Power Series Solutions of Differential Equations

The command to solve differential equations in power series around a particular initial point with specific initial conditions is called **seriesSolve**. It can take a variety of parameters, so we illustrate its use with some examples.

Since the coefficients of some solutions are quite large, we reset the default to compute only seven terms.

```
)set streams calculate 7
```

You can solve a single nonlinear equation of any order. For example, we solve

$$y''' = sin(y'') * exp(y) + cos(x)$$

subject to

$$y(0) = 1, y'(0) = 0, y''(0) = 0$$

We first tell Axiom that the symbol $'y$ denotes a new operator.

```
y := operator 'y
```

$$y$$

                                            Type: BasicOperator

Enter the differential equation using y like any system function.

```
eq := D(y(x), x, 3) - sin(D(y(x), x, 2))*exp(y(x)) = cos(x)
```

$$y''' (x) - e^{y(x)} \sin(y'' (x)) = \cos(x)$$

                                    Type: Equation Expression Integer

Solve it around $x = 0$ with the initial conditions $y(0) = 1, y'(0) = y''(0) = 0$.

```
seriesSolve(eq, y, x = 0, [1, 0, 0])
```

$$1 + \frac{1}{6} x^3 + \frac{e}{24} x^4 + \frac{e^2 - 1}{120} x^5 + \frac{e^3 - 2\,e}{720} x^6 + \frac{e^4 - 8\,e^2 + 4\,e + 1}{5040} x^7 + O\left(x^8\right)$$

                    Type: UnivariateTaylorSeries(Expression Integer,x,0)

You can also solve a system of nonlinear first order equations. For example, we solve a system that has $tan(t)$ and $sec(t)$ as solutions.

We tell Axiom that x is also an operator.

```
x := operator 'x
```

$$x$$

<div align="right">Type: BasicOperator</div>

Enter the two equations forming our system.

```
eq1 := D(x(t), t) = 1 + x(t)**2
```

$$x' \left(t \right) = x \left(t \right)^2 + 1$$

<div align="right">Type: Equation Expression Integer</div>

```
eq2 := D(y(t), t) = x(t) * y(t)
```

$$y' \left(t \right) = x \left(t \right) y \left(t \right)$$

<div align="right">Type: Equation Expression Integer</div>

Solve the system around $t = 0$ with the initial conditions $x(0) = 0$ and $y(0) = 1$. Notice that since we give the unknowns in the order $[x, y]$, the answer is a list of two series in the order

$$[\text{series for } x(t), \ \text{series for } y(t)]$$

```
seriesSolve([eq2, eq1], [x, y], t = 0, [y(0) = 1, x(0) = 0])
```

    Compiling function %BZ with type List UnivariateTaylorSeries(
       Expression Integer,t,0) -> UnivariateTaylorSeries(Expression
       Integer,t,0)
    Compiling function %CA with type List UnivariateTaylorSeries(
       Expression Integer,t,0) -> UnivariateTaylorSeries(Expression
       Integer,t,0)

$$\left[t + \frac{1}{3} \ t^3 + \frac{2}{15} \ t^5 + \frac{17}{315} \ t^7 + O\left(t^8 \right), 1 + \frac{1}{2} \ t^2 + \frac{5}{24} \ t^4 + \frac{61}{720} \ t^6 + O\left(t^8 \right) \right]$$

<div align="right">Type: List UnivariateTaylorSeries(Expression Integer,t,0)</div>

The order in which we give the equations and the initial conditions has no effect on the order of the solution.

8.11 Finite Fields

A *finite field* (also called a *Galois field*) is a finite algebraic structure where one can add, multiply and divide under the same laws (for example, commutativity, associativity or distributivity) as apply to the rational, real or complex numbers. Unlike those three fields, for any finite field there exists a positive prime integer p, called the **characteristic**, such that $px = 0$ for any element x in the finite field. In fact, the number of elements in a finite field is a power of the characteristic and for each prime p and positive integer n there exists exactly one finite field with p^n elements, up to isomorphism.[1]

When $n = 1$, the field has p elements and is called a *prime field*, discussed in the next section. There are several ways of implementing extensions of finite fields, and Axiom provides quite a bit of freedom to allow you to choose the one that is best for your application. Moreover, we provide operations for converting among the different representations of extensions and different extensions of a single field. Finally, note that you usually need to package-call operations from finite fields if the operations do not take as an argument an object of the field. See for more information on package-calling.

8.11.1 Modular Arithmetic and Prime Fields

Let n be a positive integer. It is well known that you can get the same result if you perform addition, subtraction or multiplication of integers and then take the remainder on dividing by n as if you had first done such remaindering on the operands, performed the arithmetic and then (if necessary) done remaindering again. This allows us to speak of arithmetic *modulo n* or, more simply *mod n*.

In Axiom, you use `IntegerMod` to do such arithmetic.

```
(a,b) :  IntegerMod 12
```

                                        Type: Void

```
(a, b) := (16, 7)
```

                        7

                            Type: IntegerMod 12

---

[1]For more information about the algebraic structure and properties of finite fields, see, for example, S. Lang, *Algebra*, Second Edition, New York: Addison-Wesley Publishing Company, Inc., 1984, ISBN 0 201 05487 6; or R. Lidl, H. Niederreiter, *Finite Fields*, Encyclopedia of Mathematics and Its Applications, Vol. 20, Cambridge: Cambridge Univ. Press, 1983, ISBN 0 521 30240 4.

```
[a - b, a * b]
```

$$[9, 4]$$

<div align="right">Type: List IntegerMod 12</div>

If n is not prime, there is only a limited notion of reciprocals and division.

```
a / b
```

> There are 12 exposed and 13 unexposed library operations named /
>   having 2 argument(s) but none was determined to be applicable.
>   Use HyperDoc Browse, or issue
>   )display op /
>   to learn more about the available operations. Perhaps
>   package-calling the operation or using coercions on the arguments
>   will allow you to apply the operation.
>
> Cannot find a definition or applicable library operation named /
>   with argument type(s)
>   IntegerMod 12
>   IntegerMod 12
>
>   Perhaps you should use "@" to indicate the required return type,
>   or "$" to specify which version of the function you need.

```
recip a
```

<div align="center">"failed"</div>

<div align="right">Type: Union("failed",...)</div>

Here 7 and 12 are relatively prime, so 7 has a multiplicative inverse modulo 12.

```
recip b
```

<div align="center">7</div>

<div align="right">Type: Union(IntegerMod 12,...)</div>

If we take n to be a prime number p, then taking inverses and, therefore, division are generally defined.

Use `PrimeField` instead of `IntegerMod` for n prime.

```
c :  PrimeField 11 := 8
```

$$8$$

<div align="right">Type: PrimeField 11</div>

```
inv c
```

$$7$$

<div align="right">Type: PrimeField 11</div>

You can also use $1/c$ and $c * *(-1)$ for the inverse of c.

```
9/c
```

$$8$$

<div align="right">Type: PrimeField 11</div>

`PrimeField` (abbreviation `PF`) checks if its argument is prime when you try to use an operation from it. If you know the argument is prime (particularly if it is large), `InnerPrimeField` (abbreviation `IPF`) assumes the argument has already been verified to be prime. If you do use a number that is not prime, you will eventually get an error message, most likely a division by zero message. For computer science applications, the most important finite fields are `PrimeField 2` and its extensions.

In the following examples, we work with the finite field with $p = 101$ elements.

```
GF101 := PF 101
```

<div align="center">PrimeField 101</div>

<div align="right">Type: Domain</div>

Like many domains in Axiom, finite fields provide an operation for returning a random element of the domain.

```
x := random()$GF101
```

$$8$$

<div align="right">Type: PrimeField 101</div>

```
y :  GF101 := 37
```

$$37$$

<div align="right">Type: PrimeField 101</div>

```
z := x/y
```

$$63$$

<div align="right">Type: PrimeField 101</div>

```
z * y - x
```

$$0$$

<div align="right">Type: PrimeField 101</div>

The element 2 is a *primitive element* of this field,

```
pe := primitiveElement()$GF101
```

$$2$$

<div align="right">Type: PrimeField 101</div>

in the sense that its powers enumerate all nonzero elements.

```
[pe**i for i in 0..99]
```

$$[1, 2, 4, 8, 16, 32, 64, 27, 54, 7, 14, 28, 56, 11, 22, 44, 88, 75, 49, 98,$$
$$95, 89, 77, 53, 5, 10, 20, 40, 80, 59, 17, 34, 68, 35, 70, 39, 78, 55, 9,$$
$$18, 36, 72, 43, 86, 71, 41, 82, 63, 25, 50, 100, 99, 97, 93, 85, 69, 37,$$
$$74, 47, 94, 87, 73, 45, 90, 79, 57, 13, 26, 52, 3, 6, 12, 24, 48, 96, 91,$$
$$81, 61, 21, 42, 84, 67, 33, 66, 31, 62, 23, 46, 92, 83, 65, 29, 58, 15, 30,$$
$$60, 19, 38, 76, 51]$$

<div align="right">Type: List PrimeField 101</div>

If every nonzero element is a power of a primitive element, how do you determine what the exponent is? Use **discreteLog**.

```
ex := discreteLog(y)
```

$$56$$

<div align="right">Type: PositiveInteger</div>

```
pe ** ex
```

$$37$$

<div align="right">Type: PrimeField 101</div>

The **order** of a nonzero element x is the smallest positive integer t such $x^t = 1$.

```
order y
```

$$25$$

<div align="right">Type: PositiveInteger</div>

The order of a primitive element is the defining $p - 1$.

```
order pe
```

$$100$$

<div align="right">Type: PositiveInteger</div>

8.11.2 Extensions of Finite Fields

When you want to work with an extension of a finite field in Axiom, you have three choices to make:

1. Do you want to generate an extension of the prime field (for example, `PrimeField 2`) or an extension of a given field?

2. Do you want to use a representation that is particularly efficient for multiplication, exponentiation and addition but uses a lot of computer memory (a representation that models the cyclic group structure of the multiplicative group of the field extension and uses a Zech logarithm table), one that uses a normal basis for the vector space structure of the field extension, or one that performs arithmetic modulo an irreducible polynomial? The cyclic group representation is only usable up to "medium" (relative to your machine's performance) sized fields. If the field is large and the normal basis is relatively simple, the normal basis representation is more efficient for exponentiation than the irreducible polynomial representation.

3. Do you want to provide a polynomial explicitly, a root of which "generates" the extension in one of the three senses in (2), or do you wish to have the polynomial generated for you?

This illustrates one of the most important features of Axiom: you can choose exactly the right data-type and representation to suit your application best.

We first tell you what domain constructors to use for each case above, and then give some examples.

Constructors that automatically generate extensions of the prime field:
```
FiniteField
FiniteFieldCyclicGroup
FiniteFieldNormalBasis
```

Constructors that generate extensions of an arbitrary field:
```
FiniteFieldExtension
FiniteFieldExtensionByPolynomial
FiniteFieldCyclicGroupExtension
FiniteFieldCyclicGroupExtensionByPolynomial
FiniteFieldNormalBasisExtension
FiniteFieldNormalBasisExtensionByPolynomial
```

Constructors that use a cyclic group representation:
```
FiniteFieldCyclicGroup
FiniteFieldCyclicGroupExtension
FiniteFieldCyclicGroupExtensionByPolynomial
```

Constructors that use a normal basis representation:
```
FiniteFieldNormalBasis
FiniteFieldNormalBasisExtension
FiniteFieldNormalBasisExtensionByPolynomial
```

Constructors that use an irreducible modulus polynomial representation:
```
FiniteField
FiniteFieldExtension
FiniteFieldExtensionByPolynomial
```

Constructors that generate a polynomial for you:
```
FiniteField
```

```
FiniteFieldExtension
FiniteFieldCyclicGroup
FiniteFieldCyclicGroupExtension
FiniteFieldNormalBasis
FiniteFieldNormalBasisExtension
```

Constructors for which you provide a polynomial:

```
FiniteFieldExtensionByPolynomial
FiniteFieldCyclicGroupExtensionByPolynomial
FiniteFieldNormalBasisExtensionByPolynomial
```

These constructors are discussed in the following sections where we collect together descriptions of extension fields that have the same underlying representation.[2]

If you don't really care about all this detail, just use `FiniteField`. As your knowledge of your application and its Axiom implementation grows, you can come back and choose an alternative constructor that may improve the efficiency of your code. Note that the exported operations are almost the same for all constructors of finite field extensions and include the operations exported by `PrimeField`.

8.11.3   Irreducible Modulus Polynomial Representations

All finite field extension constructors discussed in this section use a representation that performs arithmetic with univariate (one-variable) polynomials modulo an irreducible polynomial. This polynomial may be given explicitly by you or automatically generated. The ground field may be the prime field or one you specify. See 8.11.2 on page 417 for general information about finite field extensions.

For `FiniteField` (abbreviation `FF`) you provide a prime number p and an extension degree n. This degree can be 1.

Axiom uses the prime field `PrimeField(p)`, here `PrimeField 2`, and it chooses an irreducible polynomial of degree n, here 12, over the ground field.

```
GF4096 := FF(2,12);
```

                                                        Type: Domain

The objects in the generated field extension are polynomials of degree at most $n - 1$ with coefficients in the prime field. The polynomial indeterminate is automatically chosen by Axiom and is typically something like %A or %D.

---

[2]For more information on the implementation aspects of finite fields, see J. Grabmeier, A. Scheerhorn, *Finite Fields in AXIOM,* Technical Report, IBM Heidelberg Scientific Center, 1992.

These (strange) variables are *only* for output display; there are several ways to construct elements of this field.

The operation **index** enumerates the elements of the field extension and accepts as argument the integers from 1 to p^n.

The expression $index(p)$ always gives the indeterminate.

```
a := index(2)$GF4096
```

$$\%A$$

$$\text{Type: FiniteField(2,12)}$$

You can build polynomials in a and calculate in $GF4096$.

```
b := a**12 - a**5 + a
```

$$\%A^5 + \%A^3 + \%A + 1$$

$$\text{Type: FiniteField(2,12)}$$

```
b ** 1000
```

$$\%A^{10} + \%A^9 + \%A^7 + \%A^5 + \%A^4 + \%A^3 + \%A$$

$$\text{Type: FiniteField(2,12)}$$

```
c := a/b
```

$$\%A^{11} + \%A^8 + \%A^7 + \%A^5 + \%A^4 + \%A^3 + \%A^2$$

$$\text{Type: FiniteField(2,12)}$$

Among the available operations are **norm** and **trace**.

```
norm c
```

$$1$$

$$\text{Type: PrimeField 2}$$

```
trace c
```

$$0$$

<div align="right">Type: PrimeField 2</div>

Since any nonzero element is a power of a primitive element, how do we discover what the exponent is?

The operation **discreteLog** calculates the exponent and, if it is called with only one argument, always refers to the primitive element returned by **primitiveElement**.

```
dL := discreteLog a
```

$$1729$$

<div align="right">Type: PositiveInteger</div>

```
g ** dL
```

$$g^{1729}$$

<div align="right">Type: Polynomial Integer</div>

`FiniteFieldExtension` (abbreviation `FFX`) is similar to `FiniteField` except that the ground-field for `FiniteFieldExtension` is arbitrary and chosen by you.

In case you select the prime field as ground field, there is essentially no difference between the constructed two finite field extensions.

```
GF16 := FF(2,4);
```

<div align="right">Type: Domain</div>

```
GF4096 := FFX(GF16,3);
```

<div align="right">Type: Domain</div>

```
r := (random()$GF4096) ** 20
```

$$\left(\%B^2 + 1\right)\%C^2 + \left(\%B^3 + \%B^2 + 1\right)\%C + \%B^3 + \%B^2 + \%B + 1$$

<div align="center">

Type: FiniteFieldExtension(FiniteField(2,4),3)

</div>

```
norm(r)
```

$$\%B^2 + \%B$$

<div align="right">

Type: FiniteField(2,4)

</div>

FiniteFieldExtensionByPolynomial (abbreviation **FFP**) is similar to
FiniteField and **FiniteFieldExtension** but is more general.

```
GF4 := FF(2,2);
```

<div align="right">

Type: Domain

</div>

```
f := nextIrreduciblePoly(random(6)$FFPOLY(GF4))$FFPOLY(GF4)
```

$$?^6 + (\%D + 1)\ ?^5 + (\%D + 1)\ ?^4 + (\%D + 1)\ ? + 1$$

<div align="center">

Type: Union(SparseUnivariatePolynomial FiniteField(2,2),...)

</div>

For **FFP** you choose both the ground field and the irreducible polynomial used in
the representation. The degree of the extension is the degree of the polynomial.

```
GF4096 := FFP(GF4,f);
```

<div align="right">

Type: Domain

</div>

```
discreteLog random()$GF4096
```

<div align="center">

582

</div>

<div align="right">

Type: PositiveInteger

</div>

8.11.4 Cyclic Group Representations

In every finite field there exist elements whose powers are all the nonzero elements of the field. Such an element is called a *primitive element*.

In `FiniteFieldCyclicGroup` (abbreviation `FFCG`) the nonzero elements are represented by the powers of a fixed primitive element of the field (that is, a generator of its cyclic multiplicative group). Multiplication (and hence exponentiation) using this representation is easy. To do addition, we consider our primitive element as the root of a primitive polynomial (an irreducible polynomial whose roots are all primitive). See 8.11.7 on page 431 for examples of how to compute such a polynomial.

To use `FiniteFieldCyclicGroup` you provide a prime number and an extension degree.

```
GF81 := FFCG(3,4);
```

<div align="right">Type: Domain</div>

Axiom uses the prime field, here `PrimeField 3`, as the ground field and it chooses a primitive polynomial of degree n, here 4, over the prime field.

```
a := primitiveElement()$GF81
```

$$\%F^1$$

<div align="right">Type: FiniteFieldCyclicGroup(3,4)</div>

You can calculate in $GF81$.

```
b := a**12 - a**5 + a
```

$$\%F^{72}$$

<div align="right">Type: FiniteFieldCyclicGroup(3,4)</div>

In this representation of finite fields the discrete logarithm of an element can be seen directly in its output form.

```
b
```

$$\%F^{72}$$

```
                               Type:   FiniteFieldCyclicGroup(3,4)
```

```
discreteLog b
```

$$72$$

```
                                       Type:   PositiveInteger
```

FiniteFieldCyclicGroupExtension (abbreviation **FFCGX**) is simi-
lar to **FiniteFieldCyclicGroup** except that the ground field for
FiniteFieldCyclicGroupExtension is arbitrary and chosen by you.   In
case you select the prime field as ground field, there is essentially no difference
between the constructed two finite field extensions.

```
GF9 := FF(3,2);
```

```
                                            Type:   Domain
```

```
GF729 := FFCGX(GF9,3);
```

```
                                            Type:   Domain
```

```
r := (random()$GF729) ** 20
```

$$\%H^{420}$$

```
        Type:   FiniteFieldCyclicGroupExtension(FiniteField(3,2),3)
```

```
trace(r)
```

$$0$$

```
                                       Type:   FiniteField(3,2)
```

FiniteFieldCyclicGroupExtensionByPolynomial (abbreviation **FFCGP**) is
similar to **FiniteFieldCyclicGroup** and **FiniteFieldCyclicGroupExtension**
but is more general. For **FiniteFieldCyclicGroupExtensionByPolynomial**
you choose both the ground field and the irreducible polynomial used in the
representation. The degree of the extension is the degree of the polynomial.

```
GF3 := PrimeField 3;
```

<div align="right">Type: Domain</div>

We use a utility operation to generate an irreducible primitive polynomial (see 8.11.7 on page 431). The polynomial has one variable that is "anonymous": it displays as a question mark.

```
f := createPrimitivePoly(4)$FFPOLY(GF3)
```

$$?^4 + ? + 2$$

<div align="center">Type: SparseUnivariatePolynomial PrimeField 3</div>

```
GF81 := FFCGP(GF3,f);
```

<div align="right">Type: Domain</div>

Let's look at a random element from this field.

```
random()$GF81
```

$$\%K^{13}$$

<div align="center">Type: FiniteFieldCyclicGroupExtensionByPolynomial(PrimeField
3,?**4+?+2)</div>

8.11.5 Normal Basis Representations

Let K be a finite extension of degree n of the finite field F and let F have q elements. An element x of K is said to be *normal* over F if the elements

$$1, x^q, x^{q^2}, \ldots, x^{q^{n-1}}$$

form a basis of K as a vector space over F. Such a basis is called a *normal basis*.[3]

If x is normal over F, its minimal polynomial is also said to be *normal* over F. There exist normal bases for all finite extensions of arbitrary finite fields.

In `FiniteFieldNormalBasis` (abbreviation `FFNB`), the elements of the finite field are represented by coordinate vectors with respect to a normal basis.

You provide a prime p and an extension degree n.

---

[3]This agrees with the general definition of a normal basis because the n distinct powers of the automorphism $x \mapsto x^q$ constitute the Galois group of K/F.

```
K := FFNB(3,8)
```

$$FiniteFieldNormalBasis(3,8)$$

<div align="right">Type: Domain</div>

Axiom uses the prime field `PrimeField(p)`, here `PrimeField 3`, and it chooses a normal polynomial of degree n, here 8, over the ground field. The remainder class of the indeterminate is used as the normal element. The polynomial indeterminate is automatically chosen by Axiom and is typically something like $\%A$ or $\%D$. These (strange) variables are only for output display; there are several ways to construct elements of this field. The output of the basis elements is something like $\%A^{q^i}$.

```
a := normalElement()$K
```

$$\%I$$

<div align="right">Type: FiniteFieldNormalBasis(3,8)</div>

You can calculate in K using a.

```
b := a**12 - a**5 + a
```

$$2 \; \%I^{q^7} + \%I^{q^5} + \%I^q$$

<div align="right">Type: FiniteFieldNormalBasis(3,8)</div>

`FiniteFieldNormalBasisExtension`   (abbreviation   `FFNBX`)   is   similar   to   `FiniteFieldNormalBasis`   except   that   the   groundfield   for `FiniteFieldNormalBasisExtension` is arbitrary and chosen by you.   In case you select the prime field as ground field, there is essentially no difference between the constructed two finite field extensions.

```
GF9 := FFNB(3,2);
```

<div align="right">Type: Domain</div>

```
GF729 := FFNBX(GF9,3);
```

<div align="right">Type: Domain</div>

```
r := random()$GF729
```

$$2 \ \%K \ \%L^q$$

```
r + r**3 + r**9 + r**27
```

$$2 \ \%K \ \%L^{q^2} + (2 \ \%K^q + 2 \ \%K) \ \%L^q + 2 \ \%K^q \ \%L$$

`FiniteFieldNormalBasisExtensionByPolynomial` (abbreviation FFNBP) is
similar to `FiniteFieldNormalBasis` and `FiniteFieldNormalBasisExtension`
but is more general. For `FiniteFieldNormalBasisExtensionByPolynomial`
you choose both the ground field and the irreducible polynomial used in the
representation. The degree of the extension is the degree of the polynomial.

```
GF3 := PrimeField 3;
```

                                                   Type: Domain

We use a utility operation to generate an irreducible normal polynomial (see
8.11.7 on page 431). The polynomial has one variable that is "anonymous": it
displays as a question mark.

```
f := createNormalPoly(4)$FFPOLY(GF3)
```

$$?^4 + 2 \ ?^3 + 2$$

                 Type: SparseUnivariatePolynomial PrimeField 3

```
GF81 := FFNBP(GF3,f);
```

                                                   Type: Domain

Let's look at a random element from this field.

```
r := random()$GF81
```

$$\%M^{q^2} + 2\ \%M^q + 2\ \%M$$

> Type: FiniteFieldNormalBasisExtensionByPolynomial(PrimeField
> 3,?**4+2*?**3+2)

```
r * r**3 * r**9 * r**27
```

$$2\ \%M^{q^3} + 2\ \%M^{q^2} + 2\ \%M^q + 2\ \%M$$

> Type: FiniteFieldNormalBasisExtensionByPolynomial(PrimeField
> 3,?**4+2*?**3+2)

```
norm r
```

$$2$$

> Type: PrimeField 3

8.11.6   Conversion Operations for Finite Fields

Let K be a finite field.

```
K := PrimeField 3
```

$$\text{PrimeField 3}$$

> Type: Domain

An extension field K_m of degree m over K is a subfield of an extension field K_n of degree n over K if and only if m divides n.

$$
\begin{array}{c}
K_n \\
| \\
K_m \quad \Longleftrightarrow \quad m|n \\
| \\
K
\end{array}
$$

FiniteFieldHomomorphisms provides conversion operations between different extensions of one fixed finite ground field and between different representations of these finite fields.

Let's choose m and n,

```
(m,n) := (4,8)
```

$$8$$

<div align="right">Type: PositiveInteger</div>

build the field extensions,

```
Km := FiniteFieldExtension(K,m)
```

$$\text{FiniteFieldExtension(PrimeField 3,4)}$$

<div align="right">Type: Domain</div>

and pick two random elements from the smaller field.

```
Kn := FiniteFieldExtension(K,n)
```

$$\text{FiniteFieldExtension(PrimeField 3,8)}$$

<div align="right">Type: Domain</div>

```
a1 := random()$Km
```

$$2\ \%A^3 + \%A^2$$

<div align="right">Type: FiniteFieldExtension(PrimeField 3,4)</div>

```
b1 := random()$Km
```

$$\%A^3 + \%A^2 + 2\ \%A + 1$$

<div align="right">Type: FiniteFieldExtension(PrimeField 3,4)</div>

Since m divides n, K_m is a subfield of K_n.

```
a2 := a1 ::  Kn
```

$$\%B^4$$

<div align="right">Type: FiniteFieldExtension(PrimeField 3,8)</div>

Therefore we can convert the elements of K_m into elements of K_n.

```
b2 := b1 ::  Kn
```

$$2 \ \%B^6 + 2 \ \%B^4 + \%B^2 + 1$$

                    Type: FiniteFieldExtension(PrimeField 3,8)

To check this, let's do some arithmetic.

```
a1+b1 - ((a2+b2) ::  Km)
```

$$0$$

                    Type: FiniteFieldExtension(PrimeField 3,4)

```
a1*b1 - ((a2*b2) ::  Km)
```

$$0$$

                    Type: FiniteFieldExtension(PrimeField 3,4)

There are also conversions available for the situation, when K_m and K_n are represented in different ways (see 8.11.2 on page 417). For example let's choose K_m where the representation is 0 plus the cyclic multiplicative group and K_n with a normal basis representation.

```
Km := FFCGX(K,m)
```

FiniteFieldCyclicGroupExtension(PrimeField 3,4)

                                        Type: Domain

```
Kn := FFNBX(K,n)
```

FiniteFieldNormalBasisExtension(PrimeField 3,8)

                                        Type: Domain

```
(a1,b1) := (random()$Km,random()$Km)
```

$$\%C^{13}$$

        Type: FiniteFieldCyclicGroupExtension(PrimeField 3,4)

a2 := a1 :: Kn

$$2 \ \%D^{q^6} + 2 \ \%D^{q^5} + 2 \ \%D^{q^4} + 2 \ \%D^{q^2} + 2 \ \%D^q + 2 \ \%D$$

        Type: FiniteFieldNormalBasisExtension(PrimeField 3,8)

b2 := b1 :: Kn

$$2 \ \%D^{q^7} + \%D^{q^6} + \%D^{q^5} + \%D^{q^4} + 2 \ \%D^{q^3} + \%D^{q^2} + \%D^q + \%D$$

        Type: FiniteFieldNormalBasisExtension(PrimeField 3,8)

Check the arithmetic again.

a1+b1 - ((a2+b2) :: Km)

$$0$$

        Type: FiniteFieldCyclicGroupExtension(PrimeField 3,4)

a1*b1 - ((a2*b2) :: Km)

$$0$$

        Type: FiniteFieldCyclicGroupExtension(PrimeField 3,4)

8.11.7   Utility Operations for Finite Fields

`FiniteFieldPolynomialPackage` (abbreviation `FFPOLY`) provides operations
for generating, counting and testing polynomials over finite fields. Let's start
with a couple of definitions:

- A polynomial is *primitive* if its roots are primitive elements in an extension
  of the coefficient field of degree equal to the degree of the polynomial.

- A polynomial is *normal* over its coefficient field if its roots are linearly
  independent elements in an extension of the coefficient field of degree
  equal to the degree of the polynomial.

In what follows, many of the generated polynomials have one "anonymous" variable. This indeterminate is displayed as a question mark (''?'').

To fix ideas, let's use the field with five elements for the first few examples.

```
GF5 := PF 5;
```

<div align="right">Type: Domain</div>

You can generate irreducible polynomials of any (positive) degree (within the storage capabilities of the computer and your ability to wait) by using **createIrreduciblePoly**.

```
f := createIrreduciblePoly(8)$FFPOLY(GF5)
```

$$?^8 + ?^4 + 2$$

<div align="right">Type: SparseUnivariatePolynomial PrimeField 5</div>

Does this polynomial have other important properties? Use **primitive?** to test whether it is a primitive polynomial.

```
primitive?(f)$FFPOLY(GF5)
```

<div align="center">false</div>

<div align="right">Type: Boolean</div>

Use **normal?** to test whether it is a normal polynomial.

```
normal?(f)$FFPOLY(GF5)
```

<div align="center">false</div>

<div align="right">Type: Boolean</div>

Note that this is actually a trivial case, because a normal polynomial of degree n must have a nonzero term of degree $n - 1$. We will refer back to this later.

To get a primitive polynomial of degree 8 just issue this.

```
p := createPrimitivePoly(8)$FFPOLY(GF5)
```

$$?^8 + ?^3 + ?^2 + ? + 2$$

<div align="center">Type: SparseUnivariatePolynomial PrimeField 5</div>

```
primitive?(p)$FFPOLY(GF5)
```

<div align="center">true</div>

<div align="right">Type: Boolean</div>

This polynomial is not normal,

```
normal?(p)$FFPOLY(GF5)
```

<div align="center">false</div>

<div align="right">Type: Boolean</div>

but if you want a normal one simply write this.

```
n := createNormalPoly(8)$FFPOLY(GF5)
```

$$?^8 + 4\ ?^7 + ?^3 + 1$$

<div align="center">Type: SparseUnivariatePolynomial PrimeField 5</div>

This polynomial is not primitive!

```
primitive?(n)$FFPOLY(GF5)
```

<div align="center">false</div>

<div align="right">Type: Boolean</div>

This could have been seen directly, as the constant term is 1 here, which is not a primitive element up to the factor (-1) raised to the degree of the polynomial.[4]

What about polynomials that are both primitive and normal? The existence of such a polynomial is by no means obvious. [5]

If you really need one use either **createPrimitiveNormalPoly** or **createNormalPrimitivePoly**.

---

[4]Cf. Lidl, R. & Niederreiter, H., *Finite Fields,* Encycl. of Math. 20, (Addison-Wesley, 1983), p.90, Th. 3.18.

[5]The existence of such polynomials is proved in Lenstra, H. W. & Schoof, R. J., *Primitive Normal Bases for Finite Fields,* Math. Comp. 48, 1987, pp. 217-231.

```
createPrimitiveNormalPoly(8)$FFPOLY(GF5)
```

$$?^8 + 4\ ?^7 + 2\ ?^5 + 2$$

<div align="right">Type: SparseUnivariatePolynomial PrimeField 5</div>

If you want to obtain additional polynomials of the various types above as given by the **create...** operations above, you can use the **next...** operations. For instance, **nextIrreduciblePoly** yields the next monic irreducible polynomial with the same degree as the input polynomial. By "next" we mean "next in a natural order using the terms and coefficients." This will become more clear in the following examples.

This is the field with five elements.

```
GF5 := PF 5;
```

<div align="right">Type: Domain</div>

Our first example irreducible polynomial, say of degree 3, must be "greater" than this.

```
h := monomial(1,8)$SUP(GF5)
```

$$?^8$$

<div align="right">Type: SparseUnivariatePolynomial PrimeField 5</div>

You can generate it by doing this.

```
nh := nextIrreduciblePoly(h)$FFPOLY(GF5)
```

$$?^8 + 2$$

<div align="right">Type: Union(SparseUnivariatePolynomial PrimeField 5,...)</div>

Notice that this polynomial is not the same as the one **createIrreduciblePoly**.

```
createIrreduciblePoly(3)$FFPOLY(GF5)
```

$$?^3 + ? + 1$$

                    Type: SparseUnivariatePolynomial PrimeField 5

You can step through all irreducible polynomials of degree 8 over the field with
5 elements by repeatedly issuing this.

```
nh := nextIrreduciblePoly(nh)$FFPOLY(GF5)
```

$$?^8 + 3$$

          Type: Union(SparseUnivariatePolynomial PrimeField 5,...)

You could also ask for the total number of these.

```
numberOfIrreduciblePoly(5)$FFPOLY(GF5)
```

                            624

                              Type: PositiveInteger

We hope that "natural order" on polynomials is now clear: first we compare
the number of monomials of two polynomials ("more" is "greater"); then, if
necessary, the degrees of these monomials (lexicographically), and lastly their
coefficients (also lexicographically, and using the operation **lookup** if our field
is not a prime field). Also note that we make both polynomials monic before
looking at the coefficients: multiplying either polynomial by a nonzero constant
produces the same result.

The package `FiniteFieldPolynomialPackage` also provides similar operations
for primitive and normal polynomials. With the exception of the number of
primitive normal polynomials; we're not aware of any known formula for this.

```
numberOfPrimitivePoly(3)$FFPOLY(GF5)
```

                            20

                              Type: PositiveInteger

Take these,

```
m := monomial(1,1)$SUP(GF5)
```

                            ?

                    Type: SparseUnivariatePolynomial PrimeField 5

```
f := m**3 + 4*m**2 + m + 2
```

$$?^3 + 4\ ?^2 + ? + 2$$

<div align="right">Type: SparseUnivariatePolynomial PrimeField 5</div>

and then we have:

```
f1 := nextPrimitivePoly(f)$FFPOLY(GF5)
```

$$?^3 + 4\ ?^2 + 4\ ? + 2$$

<div align="right">Type: Union(SparseUnivariatePolynomial PrimeField 5,...)</div>

What happened?

```
nextPrimitivePoly(f1)$FFPOLY(GF5)
```

$$?^3 + 2\ ?^2 + 3$$

<div align="right">Type: Union(SparseUnivariatePolynomial PrimeField 5,...)</div>

Well, for the ordering used in **nextPrimitivePoly** we use as first criterion a comparison of the constant terms of the polynomials. Analogously, in **nextNormalPoly** we first compare the monomials of degree 1 less than the degree of the polynomials (which is nonzero, by an earlier remark).

```
f := m**3 + m**2 + 4*m + 1
```

$$?^3 + ?^2 + 4\ ? + 1$$

<div align="right">Type: SparseUnivariatePolynomial PrimeField 5</div>

```
f1 := nextNormalPoly(f)$FFPOLY(GF5)
```

$$?^3 + ?^2 + 4\ ? + 3$$

<div align="right">Type: Union(SparseUnivariatePolynomial PrimeField 5,...)</div>

```
nextNormalPoly(f1)$FFPOLY(GF5)
```

$$?^3 + 2\ ?^2 + 1$$

```
        Type:  Union(SparseUnivariatePolynomial PrimeField 5,...)
```

We don't have to restrict ourselves to prime fields.

Let's consider, say, a field with 16 elements.

```
GF16 := FFX(FFX(PF 2,2),2);
```

```
                                                        Type:  Domain
```

We can apply any of the operations described above.

```
createIrreduciblePoly(5)$FFPOLY(GF16)
```

$$?^5 + \%G$$

```
                            Type:  SparseUnivariatePolynomial
        FiniteFieldExtension(FiniteFieldExtension(PrimeField 2,2),2)
```

Axiom also provides operations for producing random polynomials of a given degree

```
random(5)$FFPOLY(GF16)
```

$$?^5 + (\%F\ \%G + 1)\ ?^4 + \%F\ \%G\ ?^3 + (\%G + \%F + 1)\ ?^2 +$$

$$((\%F + 1)\ \%G + \%F)\ ? + 1$$

```
                            Type:  SparseUnivariatePolynomial
        FiniteFieldExtension(FiniteFieldExtension(PrimeField 2,2),2)
```

or with degree between two given bounds.

```
random(3,9)$FFPOLY(GF16)
```

$$?^3 + (\%F\ \%G + 1)\ ?^2 + (\%G + \%F + 1)\ ? + 1$$

```
                            Type:  SparseUnivariatePolynomial
        FiniteFieldExtension(FiniteFieldExtension(PrimeField 2,2),2)
```

`FiniteFieldPolynomialPackage2` (abbreviation `FFPOLY2`) exports an opera-
tion **rootOfIrreduciblePoly** for finding one root of an irreducible polynomial
f in an extension field of the coefficient field. The degree of the extension has
to be a multiple of the degree of f. It is not checked whether f actually is
irreducible.

To illustrate this operation, we fix a ground field GF

`GF2 := PrimeField 2;`

<div align="right">Type: Domain</div>

and then an extension field.

`F := FFX(GF2,12)`

<div align="center">FiniteFieldExtension(PrimeField 2,12)</div>

<div align="right">Type: Domain</div>

We construct an irreducible polynomial over $GF2$.

`f := createIrreduciblePoly(6)$FFPOLY(GF2)`

$$?^6 + ? + 1$$

<div align="center">Type: SparseUnivariatePolynomial PrimeField 2</div>

We compute a root of f.

`root := rootOfIrreduciblePoly(f)$FFPOLY2(F,GF2)`

$$\%H^{11} + \%H^8 + \%H^7 + \%H^5 + \%H + 1$$

<div align="center">Type: FiniteFieldExtension(PrimeField 2,12)</div>

and check the result

`eval(f, monomial(1,1)$SUP(F) = root)`

$$0$$

<div align="center">Type: SparseUnivariatePolynomial FiniteFieldExtension(PrimeField
2,12)</div>

8.12 Primary Decomposition of Ideals

Axiom provides a facility for the primary decomposition of polynomial ideals over fields of characteristic zero. The algorithm works in essentially two steps:

1. the problem is solved for 0-dimensional ideals by "generic" projection on the last coordinate

2. a "reduction process" uses localization and ideal quotients to reduce the general case to the 0-dimensional one.

The Axiom constructor `PolynomialIdeals` represents ideals with coefficients in any field and supports the basic ideal operations, including intersection, sum and quotient. `IdealDecompositionPackage` contains the specific operations for the primary decomposition and the computation of the radical of an ideal with polynomial coefficients in a field of characteristic 0 with an effective algorithm for factoring polynomials.

The following examples illustrate the capabilities of this facility.

First consider the ideal generated by $x^2 + y^2 - 1$ (which defines a circle in the (x, y)-plane) and the ideal generated by $x^2 - y^2$ (corresponding to the straight lines $x = y$ and $x = -y$.

```
(n,m)  :  List DMP([x,y],FRAC INT)
```

<div align="right">Type: Void</div>

```
m  := [x**2+y**2-1]
```

$$\left[x^2 + y^2 - 1\right]$$

Type: List DistributedMultivariatePolynomial([x,y],Fraction Integer)

```
n  := [x**2-y**2]
```

$$\left[x^2 - y^2\right]$$

Type: List DistributedMultivariatePolynomial([x,y],Fraction Integer)

We find the equations defining the intersection of the two loci. This correspond to the sum of the associated ideals.

```
id := ideal m + ideal n
```

$$\left[x^2 - \frac{1}{2}, y^2 - \frac{1}{2} \right]$$

Type: PolynomialIdeals(Fraction Integer,
DirectProduct(2,NonNegativeInteger),OrderedVariableList [x,y],
DistributedMultivariatePolynomial([x,y],Fraction Integer))

We can check if the locus contains only a finite number of points, that is, if the ideal is zero-dimensional.

```
zeroDim?  id
```

true

Type: Boolean

```
zeroDim?(ideal m)
```

false

Type: Boolean

```
dimension ideal m
```

1

Type: PositiveInteger

We can find polynomial relations among the generators (f and g are the parametric equations of the knot).

```
(f,g):DMP([x,y],FRAC INT)
```

Type: Void

```
f := x**2-1
```

$$x^2 - 1$$

```
    Type:  DistributedMultivariatePolynomial([x,y],Fraction Integer)
```

```
g := x*(x**2-1)
```

$$x^3 - x$$

```
    Type:  DistributedMultivariatePolynomial([x,y],Fraction Integer)
```

```
relationsIdeal [f,g]
```

$$\left[-\%B^2 + \%A^3 + \%A^2\right] \mid \left[\%A = x^2 - 1, \%B = x^3 - x\right]$$

```
   Type:  SuchThat(List Polynomial Fraction Integer, List Equation
                                        Polynomial Fraction Integer)
```

We can compute the primary decomposition of an ideal.

```
l:  List DMP([x,y,z],FRAC INT)
```

```
                                                    Type:  Void
```

```
l:=[x**2+2*y**2,x*z**2-y*z,z**2-4]
```

$$\left[x^2 + 2\ y^2, x\ z^2 - y\ z, z^2 - 4\right]$$

```
    Type:  List DistributedMultivariatePolynomial([x,y,z],Fraction
                                                              Integer)
```

```
ld:=primaryDecomp ideal l
```

$$\left[\left[x + \frac{1}{2}\ y, y^2, z + 2\right], \left[x - \frac{1}{2}\ y, y^2, z - 2\right]\right]$$

```
                Type:  List PolynomialIdeals(Fraction Integer,
DirectProduct(3,NonNegativeInteger), OrderedVariableList [x,y,z],
    DistributedMultivariatePolynomial([x,y,z],Fraction Integer))
```

We can intersect back.

```
reduce(intersect,ld)
```

$$\left[x - \frac{1}{4} \ y \ z, y^2, z^2 - 4 \right]$$

```
                Type:  PolynomialIdeals(Fraction Integer,
DirectProduct(3,NonNegativeInteger), OrderedVariableList [x,y,z],
    DistributedMultivariatePolynomial([x,y,z],Fraction Integer))
```

We can compute the radical of every primary component.

```
reduce(intersect,[radical ld.i for i in 1..2])
```

$$\left[x, y, z^2 - 4 \right]$$

```
                Type:  PolynomialIdeals(Fraction Integer,
DirectProduct(3,NonNegativeInteger), OrderedVariableList [x,y,z],
    DistributedMultivariatePolynomial([x,y,z],Fraction Integer))
```

Their intersection is equal to the radical of the ideal of l.

```
radical ideal l
```

$$\left[x, y, z^2 - 4 \right]$$

```
                Type:  PolynomialIdeals(Fraction Integer,
DirectProduct(3,NonNegativeInteger), OrderedVariableList [x,y,z],
    DistributedMultivariatePolynomial([x,y,z],Fraction Integer))
```

8.13   Computation of Galois Groups

As a sample use of Axiom's algebraic number facilities, we compute the Galois group of the polynomial $p(x) = x^5 - 5x + 12$.

```
p := x**5 - 5*x + 12
```

$$x^5 - 5 \ x + 12$$

```
                                    Type:  Polynomial Integer
```

We would like to construct a polynomial $f(x)$ such that the splitting field of $p(x)$ is generated by one root of $f(x)$. First we construct a polynomial $r = r(x)$ such that one root of $r(x)$ generates the field generated by two roots of the

polynomial $p(x)$. (As it will turn out, the field generated by two roots of $p(x)$ is, in fact, the splitting field of $p(x)$.)

From the proof of the primitive element theorem we know that if a and b are algebraic numbers, then the field $\mathbf{Q}(a, b)$ is equal to $\mathbf{Q}(a + kb)$ for an appropriately chosen integer k. In our case, we construct the minimal polynomial of $a_i - a_j$, where a_i and a_j are two roots of $p(x)$. We construct this polynomial using **resultant**. The main result we need is the following: If $f(x)$ is a polynomial with roots $a_i \ldots a_m$ and $g(x)$ is a polynomial with roots $b_i \ldots b_n$, then the polynomial $h(x) = resultant(f(y), g(x - y), y)$ is a polynomial of degree $m * n$ with roots $a_i + b_j, i = 1 \ldots m, j = 1 \ldots n$.

For $f(x)$ we use the polynomial $p(x)$. For $g(x)$ we use the polynomial $-p(-x)$. Thus, the polynomial we first construct is $resultant(p(y), -p(y - x), y)$.

```
q := resultant(eval(p,x,y),-eval(p,x,y-x),y)
```

$x^{25} - 50\ x^{21} - 2375\ x^{17} + 90000\ x^{15} - 5000\ x^{13} + 2700000\ x^{11} + 250000\ x^{9} +$

$18000000\ x^{7} + 64000000\ x^{5}$

$$\text{Type:} \quad \text{Polynomial Integer}$$

The roots of $q(x)$ are $a_i - a_j, i \leq 1, j \leq 5$. Of course, there are five pairs (i, j) with $i = j$, so 0 is a 5-fold root of $q(x)$.

Let's get rid of this factor.

```
q1 := exquo(q, x**5)
```

$x^{20} - 50\ x^{16} - 2375\ x^{12} + 90000\ x^{10} - 5000\ x^{8} + 2700000\ x^{6} +$

$250000\ x^{4} + 18000000\ x^{2} + 64000000$

$$\text{Type:} \quad \text{Union(Polynomial Integer,...)}$$

Factor the polynomial $q1$.

```
factoredQ := factor q1
```

$\left(x^{10} - 10\ x^{8} - 75\ x^{6} + 1500\ x^{4} - 5500\ x^{2} + 16000\right) *$

$\left(x^{10} + 10\ x^{8} + 125\ x^{6} + 500\ x^{4} + 2500\ x^{2} + 4000\right)$

$$\text{Type:} \quad \text{Factored Polynomial Integer}$$

We see that $q1$ has two irreducible factors, each of degree 10. (The fact that the polynomial $q1$ has two factors of degree 10 is enough to show that the Galois group of $p(x)$ is the dihedral group of order 10.[6] Note that the type of $factoredQ$ is `FR POLY INT`, that is, `Factored Polynomial Integer`. This is a special data type for recording factorizations of polynomials with integer coefficients.

We can access the individual factors using the operation **nthFactor**.

```
r := nthFactor(factoredQ,1)
```

$$x^{10} - 10\ x^8 - 75\ x^6 + 1500\ x^4 - 5500\ x^2 + 16000$$

<div align="right">

Type: Polynomial Integer
</div>

Consider the polynomial $r = r(x)$. This is the minimal polynomial of the difference of two roots of $p(x)$. Thus, the splitting field of $p(x)$ contains a subfield of degree 10. We show that this subfield is, in fact, the splitting field of $p(x)$ by showing that $p(x)$ factors completely over this field.

First we create a symbolic root of the polynomial $r(x)$. (We replaced x by b in the polynomial r so that our symbolic root would be printed as b.)

```
beta:AN := rootOf(eval(r,x,b))
```

$$b$$

<div align="right">

Type: AlgebraicNumber
</div>

We next tell Axiom to view $p(x)$ as a univariate polynomial in x with algebraic number coefficients. This is accomplished with this type declaration.

```
p := p::UP(x,INT)::UP(x,AN)
```

$$x^5 - 5\ x + 12$$

<div align="right">

Type: UnivariatePolynomial(x,AlgebraicNumber)
</div>

Factor $p(x)$ over the field $\mathbf{Q}(\beta)$. (This computation will take some time!)

```
algFactors := factor(p,[beta])
```

---

[6]See McKay, Soicher, Computing Galois Groups over the Rationals, Journal of Number Theory 20, 273-281 (1983). We do not assume the results of this paper, however, and we continue with the computation.

$$\left(x + \dfrac{\left(\begin{array}{c} -85\ b^9 - 116\ b^8 + 780\ b^7 + 2640\ b^6 + 14895\ b^5 - \\[4pt] 8820\ b^4 - 127050\ b^3 - 327000\ b^2 - 405200\ b + 2062400 \end{array} \right)}{1339200} \right)$$

$$\left(x + \dfrac{-17\ b^8 + 156\ b^6 + 2979\ b^4 - 25410\ b^2 - 14080}{66960} \right)$$

$$\left(x + \dfrac{143\ b^8 - 2100\ b^6 - 10485\ b^4 + 290550\ b^2 - 334800\ b - 960800}{669600} \right)$$

$$\left(x + \dfrac{143\ b^8 - 2100\ b^6 - 10485\ b^4 + 290550\ b^2 + 334800\ b - 960800}{669600} \right)$$

$$\left(x + \dfrac{\left(\begin{array}{c} 85\ b^9 - 116\ b^8 - 780\ b^7 + 2640\ b^6 - 14895\ b^5 - \\[4pt] 8820\ b^4 + 127050\ b^3 - 327000\ b^2 + 405200\ b + 2062400 \end{array} \right)}{1339200} \right)$$

<div align="center">Type: Factored UnivariatePolynomial(x,AlgebraicNumber)</div>

When factoring over number fields, it is important to specify the field over which
the polynomial is to be factored, as polynomials have different factorizations
over different fields. When you use the operation **factor**, the field over which
the polynomial is factored is the field generated by

1. the algebraic numbers that appear in the coefficients of the polynomial,
   and

2. the algebraic numbers that appear in a list passed as an optional second
   argument of the operation.

In our case, the coefficients of p are all rational integers and only *beta* appears
in the list, so the field is simply $\mathbf{Q}(\beta)$.

It was necessary to give the list [*beta*] as a second argument of the operation
because otherwise the polynomial would have been factored over the field gen-
erated by its coefficients, namely the rational numbers.

```
factor(p)
```

$$x^5 - 5\ x + 12$$

```
            Type:  Factored UnivariatePolynomial(x,AlgebraicNumber)
```

We have shown that the splitting field of $p(x)$ has degree 10. Since the symmetric group of degree 5 has only one transitive subgroup of order 10, we know that the Galois group of $p(x)$ must be this group, the dihedral group of order 10. Rather than stop here, we explicitly compute the action of the Galois group on the roots of $p(x)$.

First we assign the roots of $p(x)$ as the values of five variables.

We can obtain an individual root by negating the constant coefficient of one of the factors of $p(x)$.

```
factor1 := nthFactor(algFactors,1)
```

$$x + \frac{\begin{pmatrix} -85\ b^9 - 116\ b^8 + 780\ b^7 + 2640\ b^6 + 14895\ b^5 - \\ 8820\ b^4 - 127050\ b^3 - 327000\ b^2 - 405200\ b + 2062400 \end{pmatrix}}{1339200}$$

```
                Type:  UnivariatePolynomial(x,AlgebraicNumber)
```

```
root1 := -coefficient(factor1,0)
```

$$\frac{\begin{pmatrix} 85\ b^9 + 116\ b^8 - 780\ b^7 - 2640\ b^6 - 14895\ b^5 + \\ 8820\ b^4 + 127050\ b^3 + 327000\ b^2 + 405200\ b - 2062400 \end{pmatrix}}{1339200}$$

```
                            Type:  AlgebraicNumber
```

We can obtain a list of all the roots in this way.

```
roots := [-coefficient(nthFactor(algFactors,i),0) for i in 1..5]
```

$$\left[\frac{\left(\begin{array}{c}85\ b^9 + 116\ b^8 - 780\ b^7 - 2640\ b^6 - 14895\ b^5 + 8820\ b^4 + \\ 127050\ b^3 + 327000\ b^2 + 405200\ b - 2062400\end{array}\right)}{1339200,}\right.$$

$$\frac{17\ b^8 - 156\ b^6 - 2979\ b^4 + 25410\ b^2 + 14080}{66960},$$

$$\frac{-143\ b^8 + 2100\ b^6 + 10485\ b^4 - 290550\ b^2 + 334800\ b + 960800}{669600},$$

$$\frac{-143\ b^8 + 2100\ b^6 + 10485\ b^4 - 290550\ b^2 - 334800\ b + 960800}{669600},$$

$$\left.\frac{\left(\begin{array}{c}-85\ b^9 + 116\ b^8 + 780\ b^7 - 2640\ b^6 + 14895\ b^5 + 8820\ b^4 - \\ 127050\ b^3 + 327000\ b^2 - 405200\ b - 2062400\end{array}\right)}{1339200}\right]$$

$$\text{Type:} \quad \text{List AlgebraicNumber}$$

The expression

```
- coefficient(nthFactor(algFactors, i), 0)}
```

is the i-th root of $p(x)$ and the elements of *roots* are the i-th roots of $p(x)$ as i ranges from 1 to 5.

Assign the roots as the values of the variables $a1, ..., a5$.

```
(a1,a2,a3,a4,a5) := (roots.1,roots.2,roots.3,roots.4,roots.5)
```

$$\frac{\left(\begin{array}{c}-85\ b^9 + 116\ b^8 + 780\ b^7 - 2640\ b^6 + 14895\ b^5 + 8820\ b^4 - \\ 127050\ b^3 + 327000\ b^2 - 405200\ b - 2062400\end{array}\right)}{1339200}$$

$$\text{Type:} \quad \text{AlgebraicNumber}$$

Next we express the roots of $r(x)$ as polynomials in *beta*. We could obtain these roots by calling the operation **factor**: $factor(r, [beta])$ factors $r(x)$ over $\mathbf{Q}(\beta)$. However, this is a lengthy computation and we can obtain the roots of $r(x)$ as

differences of the roots $a1, ..., a5$ of $p(x)$. Only ten of these differences are roots of $r(x)$ and the other ten are roots of the other irreducible factor of $q1$. We can determine if a given value is a root of $r(x)$ by evaluating $r(x)$ at that particular value. (Of course, the order in which factors are returned by the operation **factor** is unimportant and may change with different implementations of the operation. Therefore, we cannot predict in advance which differences are roots of $r(x)$ and which are not.)

Let's look at four examples (two are roots of $r(x)$ and two are not).

```
eval(r,x,a1 - a2)
```

$$0$$

<div align="right">Type: Polynomial AlgebraicNumber</div>

```
eval(r,x,a1 - a3)
```

$$\frac{\left(\begin{array}{l} 47905\ b^9 + 66920\ b^8 - 536100\ b^7 - 980400\ b^6 - 3345075\ b^5 - 5787000\ b^4 + \\ 75572250\ b^3 + 161688000\ b^2 - 184600000\ b - 710912000 \end{array}\right)}{4464}$$

<div align="right">Type: Polynomial AlgebraicNumber</div>

```
eval(r,x,a1 - a4)
```

$$0$$

<div align="right">Type: Polynomial AlgebraicNumber</div>

```
eval(r,x,a1 - a5)
```

$$\frac{405\ b^8 + 3450\ b^6 - 19875\ b^4 - 198000\ b^2 - 588000}{31}$$

<div align="right">Type: Polynomial AlgebraicNumber</div>

Take one of the differences that was a root of $r(x)$ and assign it to the variable *bb*.

For example, if $eval(r, x, a1 - a4)$ returned 0, you would enter this.

```
bb := a1 - a4
```

$$\frac{\left(\begin{array}{l} 85\ b^9 + 402\ b^8 - 780\ b^7 - 6840\ b^6 - 14895\ b^5 - 12150\ b^4 + \\ \\ 127050\ b^3 + 908100\ b^2 + 1074800\ b - 3984000 \end{array}\right)}{1339200}$$

<div align="right">Type: AlgebraicNumber</div>

Of course, if the difference is, in fact, equal to the root *beta*, you should choose another root of $r(x)$.

Automorphisms of the splitting field are given by mapping a generator of the field, namely *beta*, to other roots of its minimal polynomial. Let's see what happens when *beta* is mapped to *bb*.

We compute the images of the roots $a1, ..., a5$ under this automorphism:

```
aa1 := subst(a1,beta = bb)
```

$$\frac{-143\ b^8 + 2100\ b^6 + 10485\ b^4 - 290550\ b^2 + 334800\ b + 960800}{669600}$$

<div align="right">Type: AlgebraicNumber</div>

```
aa2 := subst(a2,beta = bb)
```

$$\frac{\left(\begin{array}{l} -85\ b^9 + 116\ b^8 + 780\ b^7 - 2640\ b^6 + 14895\ b^5 + 8820\ b^4 - \\ \\ 127050\ b^3 + 327000\ b^2 - 405200\ b - 2062400 \end{array}\right)}{1339200}$$

<div align="right">Type: AlgebraicNumber</div>

```
aa3 := subst(a3,beta = bb)
```

$$\frac{\left(\begin{array}{l} 85\ b^9 + 116\ b^8 - 780\ b^7 - 2640\ b^6 - 14895\ b^5 + 8820\ b^4 + \\ \\ 127050\ b^3 + 327000\ b^2 + 405200\ b - 2062400 \end{array}\right)}{1339200}$$

<div align="right">Type: AlgebraicNumber</div>

```
aa4 := subst(a4,beta = bb)
```

$$\frac{-143\ b^8 + 2100\ b^6 + 10485\ b^4 - 290550\ b^2 - 334800\ b + 960800}{669600}$$

                                        Type: AlgebraicNumber


```
aa5 := subst(a5,beta = bb)
```

$$\frac{17\ b^8 - 156\ b^6 - 2979\ b^4 + 25410\ b^2 + 14080}{66960}$$

                                        Type: AlgebraicNumber

Of course, the values $aa1, ..., aa5$ are simply a permutation of the values $a1, ..., a5$.

Let's find the value of $aa1$ (execute as many of the following five commands as necessary).

```
(aa1 = a1) ::  Boolean
```

                                false

                                              Type: Boolean


```
(aa1 = a2) ::  Boolean
```

                                false

                                              Type: Boolean


```
(aa1 = a3) ::  Boolean
```

                                true

                                              Type: Boolean


```
(aa1 = a4) ::  Boolean
```

                                false

                                              Type: Boolean


```
(aa1 = a5) ::  Boolean
```

```
false
```

                                        Type: Boolean

Proceeding in this fashion, you can find the values of $aa2, ... aa5$. You have represented the automorphism $beta->bb$ as a permutation of the roots $a1, ..., a5$. If you wish, you can repeat this computation for all the roots of $r(x)$ and represent the Galois group of $p(x)$ as a subgroup of the symmetric group on five letters.

Here are two other problems that you may attack in a similar fashion:

1. Show that the Galois group of $p(x) = x^4 + 2x^3 - 2x^2 - 3x + 1$ is the dihedral group of order eight. (The splitting field of this polynomial is the Hilbert class field of the quadratic field $\mathbf{Q}(\sqrt{145})$.)

2. Show that the Galois group of $p(x) = x^6 + 108$ has order 6 and is isomorphic to S_3, the symmetric group on three letters. (The splitting field of this polynomial is the splitting field of $x^3 - 2$.)

8.14 Non-Associative Algebras and Modelling Genetic Laws

Many algebraic structures of mathematics and Axiom have a multiplication operation $*$ that satisfies the associativity law $a * (b * c) = (a * b) * c$ for all a, b and c. The octonions are a well known exception. There are many other interesting non-associative structures, such as the class of Lie algebras.[7] Lie algebras can be used, for example, to analyse Lie symmetry algebras of partial differential equations. In this section we show a different application of non-associative algebras, the modelling of genetic laws.

The Axiom library contains several constructors for creating non-associative structures, ranging from the categories `Monad`, `NonAssociativeRng`, and `FramedNonAssociativeAlgebra`, to the domains `AlgebraGivenByStructuralConstants` and `GenericNonAssociativeAlgebra`. Furthermore, the package `AlgebraPackage` provides operations for analysing the structure of such algebras.[8]

Mendel's genetic laws are often written in a form like

$$Aa \times Aa = \frac{1}{4}AA + \frac{1}{2}Aa + \frac{1}{4}aa$$

---

[7]Two Axiom implementations of Lie algebras are `LieSquareMatrix` and `FreeNilpotentLie`.

[8]these aspects of the Axiom library from the paper "Computations in Algebras of Finite Rank," by Johannes Grabmeier and Robert Wisbauer, Technical Report, IBM Heidelberg Scientific Center, 1992.

The implementation of general algebras in Axiom allows us to use this as the definition for multiplication in an algebra. Hence, it is possible to study questions of genetic inheritance using Axiom. To demonstrate this more precisely, we discuss one example from a monograph of A. Wörz-Busekros, where you can also find a general setting of this theory.[9]

We assume that there is an infinitely large random mating population. Random mating of two gametes a_i and a_j gives zygotes $a_i a_j$, which produce new gametes. In classical Mendelian segregation we have $a_i a_j = \frac{1}{2} a_i + \frac{1}{2} a_j$. In general, we have

$$a_i a_j = \sum_{k=1}^{n} \gamma_{i,j}^k \; a_k.$$

The segregation rates $\gamma_{i,j}$ are the structural constants of an n-dimensional algebra. This is provided in Axiom by the constructor `AlgebraGivenByStructuralConstants` (abbreviation `ALGSC`).

Consider two coupled autosomal loci with alleles A, a, B, and b, building four different gametes $a_1 = AB, a_2 = Ab, a_3 = aB$, and $a_4 = ab$ a1 := AB, a2 := Ab, a3 := aB, and a4 := ab. The zygotes $a_i a_j$ produce gametes a_i and a_j with classical Mendelian segregation. Zygote $a_1 a_4$ undergoes transition to $a_2 a_3$ and vice versa with probability $0 \le \theta \le \frac{1}{2}$.

Define a list $[(\gamma_{i,j}^k) 1 \le k \le 4]$ of four four-by-four matrices giving the segregation rates. We use the value $1/10$ for θ.

```
segregationRates :  List SquareMatrix(4,FRAC INT) := [matrix [
[1, 1/2, 1/2, 9/20], [1/2, 0, 1/20, 0], [1/2, 1/20, 0, 0], [9/20,
0, 0, 0] ], matrix [ [0, 1/2, 0, 1/20], [1/2, 1, 9/20, 1/2], [0,
9/20, 0, 0], [1/20, 1/2, 0, 0] ], matrix [ [0, 0, 1/2, 1/20], [0,
0, 9/20, 0], [1/2, 9/20, 1, 1/2], [1/20, 0, 1/2, 0] ], matrix [
[0, 0, 0, 9/20], [0, 0, 1/20, 1/2], [0, 1/20, 0, 1/2], [9/20,
1/2, 1/2, 1] ] ]
```

$$
\left[
\begin{bmatrix}
1 & \frac{1}{2} & \frac{1}{2} & \frac{9}{20} \\
\frac{1}{2} & 0 & \frac{1}{20} & 0 \\
\frac{1}{2} & \frac{1}{20} & 0 & 0 \\
\frac{9}{20} & 0 & 0 & 0
\end{bmatrix},
\begin{bmatrix}
0 & \frac{1}{2} & 0 & \frac{1}{20} \\
\frac{1}{2} & 1 & \frac{9}{20} & \frac{1}{2} \\
0 & \frac{9}{20} & 0 & 0 \\
\frac{1}{20} & \frac{1}{2} & 0 & 0
\end{bmatrix},
\right.
$$

$$
\left.
\begin{bmatrix}
0 & 0 & \frac{1}{2} & \frac{1}{20} \\
0 & 0 & \frac{9}{20} & 0 \\
\frac{1}{2} & \frac{9}{20} & 1 & \frac{1}{2} \\
\frac{1}{20} & 0 & \frac{1}{2} & 0
\end{bmatrix},
\begin{bmatrix}
0 & 0 & 0 & \frac{9}{20} \\
0 & 0 & \frac{1}{20} & \frac{1}{2} \\
0 & \frac{1}{20} & 0 & \frac{1}{2} \\
\frac{9}{20} & \frac{1}{2} & \frac{1}{2} & 1
\end{bmatrix}
\right]
$$

---

[9]Springer Lectures Notes in Biomathematics 36, Berlin e.a. (1980). In particular, see example 1.3.

Choose the appropriate symbols for the basis of gametes,

```
gametes := ['AB,'Ab,'aB,'ab]
```

$$[AB, Ab, aB, ab]$$

Type: List OrderedVariableList [AB,Ab,aB,ab]

Define the algebra.

```
A := ALGSC(FRAC INT, 4, gametes, segregationRates)
```

AlgebraGivenByStructuralConstants(FractionInteger, 4,
[AB, Ab, aB, ab], [MATRIX, MATRIX, MATRIX, MATRIX])

Type: Domain

What are the probabilities for zygote $a_1 a_4$ to produce the different gametes?

```
a := basis()$A
```

$$[AB, Ab, aB, ab]$$

Type: Vector AlgebraGivenByStructuralConstants(Fraction
Integer,4,[AB,Ab,aB,ab], [MATRIX,MATRIX,MATRIX,MATRIX])

```
a.1*a.4
```

$$\frac{9}{20}\ ab + \frac{1}{20}\ aB + \frac{1}{20}\ Ab + \frac{9}{20}\ AB$$

Type: AlgebraGivenByStructuralConstants(Fraction
Integer,4,[AB,Ab,aB,ab], [MATRIX,MATRIX,MATRIX,MATRIX])

Elements in this algebra whose coefficients sum to one play a distinguished role. They represent a population with the distribution of gametes reflected by the coefficients with respect to the basis of gametes.

Random mating of different populations x and y is described by their product $x * y$.

This product is commutative only if the gametes are not sex-dependent, as in our example.

```
commutative?()$A
```

$$true$$

$$Type: Boolean$$

In general, it is not associative.

```
associative?()$A
```

$$false$$

$$Type: Boolean$$

Random mating within a population x is described by $x*x$. The next generation is $(x*x)*(x*x)$.

Use decimal numbers to compare the distributions more easily.

```
x :  ALGSC(DECIMAL, 4, gametes, segregationRates) := convert
[3/10, 1/5, 1/10, 2/5]
```

$$0.4\ ab + 0.1\ aB + 0.2\ Ab + 0.3\ AB$$

```
                                                            Type:
AlgebraGivenByStructuralConstants(DecimalExpansion,4,[AB,Ab,aB,ab],
                                [MATRIX,MATRIX,MATRIX,MATRIX])
```

To compute directly the gametic distribution in the fifth generation, we use **plenaryPower**.

```
plenaryPower(x,5)
```

$$0.36561\ ab + 0.13439\ aB + 0.23439\ Ab + 0.26561\ AB$$

```
                                                            Type:
AlgebraGivenByStructuralConstants(DecimalExpansion,4,[AB,Ab,aB,ab],
                                [MATRIX,MATRIX,MATRIX,MATRIX])
```

We now ask two questions: Does this distribution converge to an equilibrium state? What are the distributions that are stable?

This is an invariant of the algebra and it is used to answer the first question. The new indeterminates describe a symbolic distribution.

```
q := leftRankPolynomial()$GCNAALG(FRAC INT, 4, gametes,
segregationRates) ::  UP(Y, POLY FRAC INT)
```

$$Y^3 + \left(-\frac{29}{20} \ \%x4 - \frac{29}{20} \ \%x3 - \frac{29}{20} \ \%x2 - \frac{29}{20} \ \%x1\right) Y^2 +$$

$$\left(\begin{array}{l} \left(\frac{9}{20} \ \%x4^2 + \left(\frac{9}{10} \ \%x3 + \frac{9}{10} \ \%x2 + \frac{9}{10} \ \%x1\right) \%x4 + \\[2ex] \dfrac{9}{20} \ \%x3^2 + \left(\dfrac{9}{10} \ \%x2 + \dfrac{9}{10} \ \%x1\right) \%x3 + \dfrac{9}{20} \ \%x2^2 + \\[2ex] \dfrac{9}{10} \ \%x1 \ \%x2 + \dfrac{9}{20} \ \%x1^2\right) \end{array}\right) Y$$

<div align="center">Type: UnivariatePolynomial(Y,Polynomial Fraction Integer)</div>

Because the coefficient $\frac{9}{20}$ has absolute value less than 1, all distributions do converge, by a theorem of this theory.

```
factor(q ::  POLY FRAC INT)
```

$$(Y - \%x4 - \%x3 - \%x2 - \%x1) *$$

$$\left(Y - \frac{9}{20} \ \%x4 - \frac{9}{20} \ \%x3 - \frac{9}{20} \ \%x2 - \frac{9}{20} \ \%x1\right) Y$$

<div align="center">Type: Factored Polynomial Fraction Integer</div>

The second question is answered by searching for idempotents in the algebra.

```
cI := conditionsForIdempotents()$GCNAALG(FRAC INT, 4, gametes,
segregationRates)
```

$$\left[\frac{9}{10} \ \%x1 \ \%x4 + \left(\frac{1}{10} \ \%x2 + \%x1\right) \%x3 + \%x1 \ \%x2 + \%x1^2 - \%x1,\right.$$

$$\left(\%x2 + \frac{1}{10} \ \%x1\right) \%x4 + \frac{9}{10} \ \%x2 \ \%x3 + \%x2^2 + (\%x1 - 1) \ \%x2,$$

$$\left(\%x3 + \frac{1}{10} \ \%x1\right) \%x4 + \%x3^2 + \left(\frac{9}{10} \ \%x2 + \%x1 - 1\right) \%x3,$$

$$\left.\%x4^2 + \left(\%x3 + \%x2 + \frac{9}{10} \ \%x1 - 1\right) \%x4 + \frac{1}{10} \ \%x2 \ \%x3\right]$$

<div align="center">Type: List Polynomial Fraction Integer</div>

Solve these equations and look at the first solution.

```
gbs:= groebnerFactorize cI
```

$$
\begin{bmatrix}
[\%x4 + \%x3 + \%x2 + \%x1 - 1, \\
\quad (\%x2 + \%x1)\ \%x3 + \%x1\ \%x2 + \%x1^2 - \%x1],
\end{bmatrix}
$$

$$
[1], [\%x4 + \%x3 - 1, \%x2, \%x1],
$$

$$
[\%x4 + \%x2 - 1, \%x3, \%x1], [\%x4, \%x3, \%x2, \%x1],
$$

$$
[\%x4 - 1, \%x3, \%x2, \%x1], \left[\%x4 - \frac{1}{2}, \%x3 - \frac{1}{2}, \%x2, \%x1\right]
$$

<div align="center">Type: List List Polynomial Fraction Integer</div>

```
gbs.1
```

$$
[\%x4 + \%x3 + \%x2 + \%x1 - 1,
$$
$$
(\%x2 + \%x1)\ \%x3 + \%x1\ \%x2 + \%x1^2 - \%x1]
$$

<div align="center">Type: List Polynomial Fraction Integer</div>

Further analysis using the package `PolynomialIdeals` shows that there is a two-dimensional variety of equilibrium states and all other solutions are contained in it.

Choose one equilibrium state by setting two indeterminates to concrete values.

```
sol := solve concat(gbs.1,[%x1-1/10,%x2-1/10])
```

$$
\left[\left[\%x4 = \frac{2}{5}, \%x3 = \frac{2}{5}, \%x2 = \frac{1}{10}, \%x1 = \frac{1}{10}\right]\right]
$$

<div align="center">Type: List List Equation Fraction Polynomial Integer</div>

```
e :  A := represents reverse (map(rhs, sol.1) ::  List FRAC INT)
```

$$
\frac{2}{5}\ ab + \frac{2}{5}\ aB + \frac{1}{10}\ Ab + \frac{1}{10}\ AB
$$

<div align="center">Type: AlgebraGivenByStructuralConstants(Fraction
Integer,4,[AB,Ab,aB,ab], [MATRIX,MATRIX,MATRIX,MATRIX])</div>

Verify the result.

```
e*e-e
```

$$0$$

```
        Type:  AlgebraGivenByStructuralConstants(Fraction
  Integer,4,[AB,Ab,aB,ab], [MATRIX,MATRIX,MATRIX,MATRIX])
```

Chapter 9

Some Examples of Domains and Packages

In this chapter we show examples of many of the most commonly used AXIOM domains and packages. The sections are organized by constructor names.

9.1 AssociationList

The `AssociationList` constructor provides a general structure for associative storage. This type provides association lists in which data objects can be saved according to keys of any type. For a given association list, specific types must be chosen for the keys and entries. You can think of the representation of an association list as a list of records with key and entry fields.

Association lists are a form of table and so most of the operations available for `Table` are also available for `AssociationList`. They can also be viewed as lists and can be manipulated accordingly.

This is a `Record` type with age and gender fields.

```
Data := Record(monthsOld :  Integer, gender :  String)
```

$$\text{Record(monthsOld: Integer,gender: String)}$$

                                                               Type: Domain

In this expression, `al` is declared to be an association list whose keys are strings and whose entries are the above records.

```
al :  AssociationList(String,Data)
```

                                                        Type: Void

The **table** operation is used to create an empty association list.

```
al := table()
```

$$table()$$

  Type: AssociationList(String,Record(monthsOld: Integer,gender:
                                                      String))

You can use assignment syntax to add things to the association list.

```
al."bob" := [407,"male"]$Data
```

$$[monthsOld = 407, gender = \texttt{"male"}]$$

                Type: Record(monthsOld: Integer,gender: String)

```
al."judith" := [366,"female"]$Data
```

$$[monthsOld = 366, gender = \texttt{"female"}]$$

                Type: Record(monthsOld: Integer,gender: String)

```
al."katie" := [24,"female"]$Data
```

$$[monthsOld = 24, gender = \texttt{"female"}]$$

                Type: Record(monthsOld: Integer,gender: String)

Perhaps we should have included a species field.

```
al."smokie" := [200,"female"]$Data
```

$$[monthsOld = 200, gender = \texttt{"female"}]$$

                Type: Record(monthsOld: Integer,gender: String)

Now look at what is in the association list. Note that the last-added (key, entry) pair is at the beginning of the list.

al

$$table\,(\texttt{"smokie"} = [monthsOld = 200, gender = \texttt{"female"}],$$

$$\texttt{"katie"} = [monthsOld = 24, gender = \texttt{"female"}],$$

$$\texttt{"judith"} = [monthsOld = 366, gender = \texttt{"female"}],$$

$$\texttt{"bob"} = [monthsOld = 407, gender = \texttt{"male"}])$$

```
Type:  AssociationList(String,Record(monthsOld:  Integer,gender:
                                                           String))
```

You can reset the entry for an existing key.

```
al."katie" := [23,"female"]$Data
```

$$[monthsOld = 23, gender = \texttt{"female"}]$$

```
Type:  Record(monthsOld:  Integer,gender:  String)
```

Use **delete** to destructively remove an element of the association list. Use **delete** to return a copy of the association list with the element deleted. The second argument is the index of the element to delete.

```
delete!(al,1)
```

$$table\,(\texttt{"katie"} = [monthsOld = 23, gender = \texttt{"female"}],$$

$$\texttt{"judith"} = [monthsOld = 366, gender = \texttt{"female"}],$$

$$\texttt{"bob"} = [monthsOld = 407, gender = \texttt{"male"}])$$

```
Type:  AssociationList(String,Record(monthsOld:  Integer,gender:
                                                           String))
```

For more information about tables, see 9.80 on page 816. For more information about lists, see 9.47 on page 675.

9.2 BalancedBinaryTree

`BalancedBinaryTrees(S)` is the domain of balanced binary trees with elements of type `S` at the nodes. A binary tree is either `empty` or else consists of a `node`

having a `value` and two branches, each branch a binary tree. A balanced binary tree is one that is balanced with respect its leaves. One with 2^k leaves is perfectly "balanced": the tree has minimum depth, and the `left` and `right` branch of every interior node is identical in shape.

Balanced binary trees are useful in algebraic computation for so-called "divide-and-conquer" algorithms. Conceptually, the data for a problem is initially placed at the root of the tree. The original data is then split into two sub-problems, one for each subtree. And so on. Eventually, the problem is solved at the leaves of the tree. A solution to the original problem is obtained by some mechanism that can reassemble the pieces. In fact, an implementation of the Chinese Remainder Algorithm using balanced binary trees was first proposed by David Y. Y. Yun at the IBM T. J. Watson Research Center in Yorktown Heights, New York, in 1978. It served as the prototype for polymorphic algorithms in Axiom.

In what follows, rather than perform a series of computations with a single expression, the expression is reduced modulo a number of integer primes, a computation is done with modular arithmetic for each prime, and the Chinese Remainder Algorithm is used to obtain the answer to the original problem. We illustrate this principle with the computation of $12^2 = 144$.

A list of moduli.

```
lm := [3,5,7,11]
```

$$[3, 5, 7, 11]$$

                                    Type: List PositiveInteger

The expression `modTree(n, lm)` creates a balanced binary tree with leaf values `n mod m` for each modulus `m` in `lm`.

```
modTree(12,lm)
```

$$[0, 2, 5, 1]$$

                                    Type: List Integer

Operation `modTree` does this using operations on balanced binary trees. We trace its steps. Create a balanced binary tree `t` of zeros with four leaves.

```
t := balancedBinaryTree(#lm, 0)
```

$$[[0, 0, 0], 0, [0, 0, 0]]$$

The leaves of the tree are set to the individual moduli.

```
setleaves!(t,lm)
```

$$[[3, 0, 5], 0, [7, 0, 11]]$$

Type: BalancedBinaryTree NonNegativeInteger

Use `mapUp!` to do a bottom-up traversal of `t`, setting each interior node to the product of the values at the nodes of its children.

```
mapUp!(t,_*)
```

$$1155$$

Type: PositiveInteger

The value at the node of every subtree is the product of the moduli of the leaves of the subtree.

```
t
```

$$[[3, 15, 5], 1155, [7, 77, 11]]$$

Type: BalancedBinaryTree NonNegativeInteger

Operation `mapDown!(t,a,fn)` replaces the value v at each node of `t` by `fn(a,v)`.

```
mapDown!(t,12,_rem)
```

$$[[0, 12, 2], 12, [5, 12, 1]]$$

Type: BalancedBinaryTree NonNegativeInteger

The operation `leaves` returns the leaves of the resulting tree. In this case, it returns the list of `12 mod m` for each modulus `m`.

```
leaves %
```

$$[0, 2, 5, 1]$$

<div align="right">Type: List NonNegativeInteger</div>

Compute the square of the images of `12` modulo each `m`.

`squares := [x**2 rem m for x in % for m in lm]`

$$[0, 4, 4, 1]$$

<div align="right">Type: List NonNegativeInteger</div>

Call the Chinese Remainder Algorithm to get the answer for 12^2.

`chineseRemainder(%,lm)`

<div align="center">144</div>

<div align="right">Type: PositiveInteger</div>

9.3   BasicOperator

A basic operator is an object that can be symbolically applied to a list of arguments from a set, the result being a kernel over that set or an expression. In addition to this section, please see 9.21 on page 540 and 9.37 on page 604 for additional information and examples.

You create an object of type `BasicOperator` by using the **operator** operation. This first form of this operation has one argument and it must be a symbol. The symbol should be quoted in case the name has been used as an identifier to which a value has been assigned.

A frequent application of `BasicOperator` is the creation of an operator to represent the unknown function when solving a differential equation.

Let `y` be the unknown function in terms of `x`.

`y := operator 'y`

<div align="center">y</div>

<div align="right">Type: BasicOperator</div>

This is how you enter the equation `y'' + y' + y = 0`.

`deq := D(y x, x, 2) + D(y x, x) + y x = 0`

$$y" \left(x \right) + y` \left(x \right) + y \left(x \right) = 0$$

<div align="right">Type: Equation Expression Integer</div>

To solve the above equation, enter this.

```
solve(deq, y, x)
```

$$\left[particular = 0, basis = \left[\cos \left(\frac{x \sqrt{3}}{2} \right) e^{\left(-\frac{x}{2} \right)}, e^{\left(-\frac{x}{2} \right)} \sin \left(\frac{x \sqrt{3}}{2} \right) \right] \right]$$

<div align="right">Type: Union(Record(particular: Expression Integer, basis: List
Expression Integer),...)</div>

See **??** on page **??** in Section **??** on page **??** for this kind of use of `BasicOperator`.

Use the single argument form of **operator** (as above) when you intend to use the operator to create functional expressions with an arbitrary number of arguments

Nary means an arbitrary number of arguments can be used in the functional expressions.

```
nary?  y
```

<div align="center">true</div>

<div align="right">Type: Boolean</div>

```
unary?  y
```

<div align="center">false</div>

<div align="right">Type: Boolean</div>

Use the two-argument form when you want to restrict the number of arguments in the functional expressions created with the operator.

This operator can only be used to create functional expressions with one argument.

```
opOne := operator('opOne, 1)
```

<div align="center">*opOne*</div>

                                        Type: BasicOperator

nary? opOne

                              false

                                            Type: Boolean

unary? opOne

                              true

                                            Type: Boolean

Use **arity** to learn the number of arguments that can be used. It returns `"false"`
if the operator is nary.

arity opOne

                              1

                              Type: Union(NonNegativeInteger,...)

Use **name** to learn the name of an operator.

name opOne

                              opOne

                                            Type: Symbol

Use **is?** to learn if an operator has a particular name.

is?(opOne, 'z2)

                              false

                                            Type: Boolean

You can also use a string as the name to be tested against.

```
is?(opOne, "opOne")
```

$$true$$

Type: Boolean

You can attached named properties to an operator. These are rarely used at the top-level of the Axiom interactive environment but are used with Axiom library source code.

By default, an operator has no properties.

```
properties y
```

$$table()$$

Type: AssociationList(String,None)

The interface for setting and getting properties is somewhat awkward because the property values are stored as values of type `None`.

Attach a property by using **setProperty**.

```
setProperty(y, "use", "unknown function" ::  None )
```

$$y$$

Type: BasicOperator

```
properties y
```

$$table\left("use" = NONE\right)$$

Type: AssociationList(String,None)

We *know* the property value has type `String`.

```
property(y, "use") ::  None pretend String
```

```
"unknown function"
```

Type: String

Use **deleteProperty!** to destructively remove a property.

```
deleteProperty!(y, "use")
```

$$y$$

                                                Type: BasicOperator

```
properties y
```

$$table()$$

                            Type: AssociationList(String,None)

9.4 BinaryExpansion

All rational numbers have repeating binary expansions. Operations to access the individual bits of a binary expansion can be obtained by converting the value to `RadixExpansion(2)`. More examples of expansions are available in 9.15 on page 529, 9.33 on page 586, and 9.65 on page 747.

The expansion (of type `BinaryExpansion`) of a rational number is returned by the **binary** operation.

```
r := binary(22/7)
```

$$11.\overline{001}$$

                                                Type: BinaryExpansion

Arithmetic is exact.

```
r + binary(6/7)
```

$$100$$

                                                Type: BinaryExpansion

The period of the expansion can be short or long . . .

```
[binary(1/i) for i in 102..106]
```

$$\left[0.0\overline{00000101}, 0.\overline{0000001001111100010001011001011110011100100101001},\right.$$

$$0.000\overline{000100111011}, 0.\overline{000000100111},$$

$$\left.0.\overline{00000010011010100100001110011111011001010110111100011}\right]$$

or very long.

```
binary(1/1007)
```

$$0.\overline{0000000001000001000101001001011110000011111100001011111100101100011111011}$$
$$\overline{000100111001001100110001100100101010111101101001100000001100001100111110}$$
$$\overline{111000110100010111101001000111101100001010111011100111010101110011001010}$$
$$\overline{010111000000011100011110010000001001001001101110010101001110100011011101}$$
$$\overline{101011100010010000011001011011000000101100101111100010100000101010101101}$$
$$\overline{011000001101101110100101011111110101110101001100100001010011011000100110}$$
$$\overline{0010001000010000110001110100111010001}$$

These numbers are bona fide algebraic objects.

```
p := binary(1/4)*x**2 + binary(2/3)*x + binary(4/9)
```

$$0.01 \; x^2 + 0.\overline{10} \; x + 0.\overline{011100}$$

```
q := D(p, x)
```

$$0.1 \; x + 0.\overline{10}$$

```
g := gcd(p, q)
```

$$x + 1.\overline{01}$$

9.5   BinarySearchTree

`BinarySearchTree(R)` is the domain of binary trees with elements of type `R`, ordered across the nodes of the tree. A non-empty binary search tree has a value of type `R`, and `right` and `left` binary search subtrees. If a subtree is empty, it is displayed as a period (".").

Define a list of values to be placed across the tree. The resulting tree has `8` at the root; all other elements are in the left subtree.

`lv := [8,3,5,4,6,2,1,5,7]`

$$[8, 3, 5, 4, 6, 2, 1, 5, 7]$$

                                    Type: List PositiveInteger

A convenient way to create a binary search tree is to apply the operation `binarySearchTree` to a list of elements.

`t := binarySearchTree lv`

$$[[[1, 2, .], 3, [4, 5, [5, 6, 7]]], 8, .]$$

                              Type: BinarySearchTree PositiveInteger

Another approach is to first create an empty binary search tree of integers.

`emptybst := empty()$BSTREE(INT)`

$$[\,]$$

                                 Type: BinarySearchTree Integer

Insert the value `8`. This establishes `8` as the root of the binary search tree. Values inserted later that are less than `8` get stored in the `left` subtree, others in the `right` subtree.

`t1 := insert!(8,emptybst)`

$$8$$

                                 Type: BinarySearchTree Integer

Insert the value 3. This number becomes the root of the **left** subtree of **t1**. For optimal retrieval, it is thus important to insert the middle elements first.

```
insert!(3,t1)
```

$$[3, 8, .]$$

                                    Type: BinarySearchTree Integer

We go back to the original tree **t**. The leaves of the binary search tree are those which have empty **left** and **right** subtrees.

```
leaves t
```

$$[1, 4, 5, 7]$$

                                    Type: List PositiveInteger

The operation **split(k,t)** returns a containing the two subtrees: one with all elements "less" than **k**, another with elements "greater" than **k**.

```
split(3,t)
```

$$[less = [1, 2, .], greater = [[., 3, [4, 5, [5, 6, 7]]], 8, .]]$$

    Type: Record(less: BinarySearchTree PositiveInteger,greater:
                                    BinarySearchTree PositiveInteger)

Define **insertRoot** to insert new elements by creating a new node.

```
insertRoot:  (INT,BSTREE INT) -> BSTREE INT
```

                                                    Type: Void

The new node puts the inserted value between its "less" tree and "greater" tree.

```
insertRoot(x, t) ==
    a := split(x, t)
    node(a.less, x, a.greater)
```

Function **buildFromRoot** builds a binary search tree from a list of elements **ls** and the empty tree **emptybst**.

```
buildFromRoot ls == reduce(insertRoot,ls,emptybst)
```

                                                      Type: Void

Apply this to the reverse of the list `lv`.

```
rt := buildFromRoot reverse lv
```

$$[[[1, 2, .], 3, [4, 5, [5, 6, 7]]], 8, .]$$

                                    Type: BinarySearchTree Integer

Have Axiom check that these are equal.

```
(t = rt)@Boolean
```

                                        true

                                              Type: Boolean

9.6   CardinalNumber

The `CardinalNumber` domain can be used for values indicating the cardinality
of sets, both finite and infinite. For example, the **dimension** operation in the
category `VectorSpace` returns a cardinal number.

The non-negative integers have a natural construction as cardinals

```
0 = #{ }, 1 = {0}, 2 = {0, 1}, ..., n = {i | 0 <= i < n}.
```

The fact that `0` acts as a zero for the multiplication of cardinals is equivalent to
the axiom of choice.

Cardinal numbers can be created by conversion from non-negative integers.

```
c0 := 0 ::  CardinalNumber
```

                                        0

                                        Type: CardinalNumber

```
c1 := 1 ::  CardinalNumber
```

$$1$$

<div align="right">Type: CardinalNumber</div>

```
c2 := 2 ::  CardinalNumber
```

$$2$$

<div align="right">Type: CardinalNumber</div>

```
c3 := 3 ::  CardinalNumber
```

$$3$$

<div align="right">Type: CardinalNumber</div>

They can also be obtained as the named cardinal `Aleph(n)`.

```
A0 := Aleph 0
```

$$Aleph\,(0)$$

<div align="right">Type: CardinalNumber</div>

```
A1 := Aleph 1
```

$$Aleph\,(1)$$

<div align="right">Type: CardinalNumber</div>

The **finite?** operation tests whether a value is a finite cardinal, that is, a non-negative integer.

```
finite?  c2
```

<div align="center">true</div>

<div align="right">Type: Boolean</div>

```
finite?  A0
```

$$false$$

Type: Boolean

Similarly, the **countable?** operation determines whether a value is a countable cardinal, that is, finite or `Aleph(0)`.

```
countable?  c2
```

$$true$$

Type: Boolean

```
countable?  A0
```

$$true$$

Type: Boolean

```
countable?  A1
```

$$false$$

Type: Boolean

Arithmetic operations are defined on cardinal numbers as follows: If `x = #X` and `y = #Y` then

$$x + y = \#(X + Y) \qquad cardinality\,of\,the\,disjoint\,union$$
$$x - y = \#(X - Y) \qquad cardinality\,of\,the\,relative\,complement$$
$$x * y = \#(X * Y) \qquad cardinality\,of\,the\,Cartesian\,product$$
$$x * {*}y = \#(X * {*}Y) \qquad cardinality\,of\,the\,set\,of\,maps\,from\,\mathtt{Y}\,to\,\mathtt{X}$$

Here are some arithmetic examples.

```
[c2 + c2, c2 + A1]
```

$$[4, Aleph\,(1)]$$

Type: List CardinalNumber

```
[c0*c2, c1*c2, c2*c2, c0*A1, c1*A1, c2*A1, A0*A1]
```

$$[0, 2, 4, 0, Aleph\,(1), Aleph\,(1), Aleph\,(1)]$$

<div align="right">Type: List CardinalNumber</div>

```
[c2**c0, c2**c1, c2**c2, A1**c0, A1**c1, A1**c2]
```

$$[1, 2, 4, 1, Aleph\,(1), Aleph\,(1)]$$

<div align="right">Type: List CardinalNumber</div>

Subtraction is a partial operation: it is not defined when subtracting a larger cardinal from a smaller one, nor when subtracting two equal infinite cardinals.

```
[c2-c1, c2-c2, c2-c3, A1-c2, A1-A0, A1-A1]
```

$$[1, 0, \texttt{"failed"}, Aleph\,(1), Aleph\,(1), \texttt{"failed"}]$$

<div align="right">Type: List Union(CardinalNumber,"failed")</div>

The generalized continuum hypothesis asserts that

```
2**Aleph i = Aleph(i+1)
```

and is independent of the axioms of set theory.[1]

The `CardinalNumber` domain provides an operation to assert whether the hypothesis is to be assumed.

```
generalizedContinuumHypothesisAssumed true
```

When the generalized continuum hypothesis is assumed, exponentiation to a transfinite power is allowed.

```
[c0**A0, c1**A0, c2**A0, A0**A0, A0**A1, A1**A0, A1**A1]
```

$$[0, 1, Aleph\,(1), Aleph\,(1), Aleph\,(2), Aleph\,(1), Aleph\,(2)]$$

<div align="right">Type: List CardinalNumber</div>

---

[1]Goedel, *The consistency of the continuum hypothesis,* Ann. Math. Studies, Princeton Univ. Press, 1940.

Three commonly encountered cardinal numbers are

$$\texttt{a} = \#\mathbf{Z} \qquad\qquad\qquad countable\ infinity$$
$$\texttt{c} = \#\mathbf{R} \qquad\qquad\qquad the\ continuum$$
$$\texttt{f} = \#\{\texttt{g}|\texttt{g}:[0,1]->\mathbf{R}\}$$

In this domain, these values are obtained under the generalized continuum hypothesis in this way.

```
a := Aleph 0
```

$$Aleph\,(0)$$

<div align="right">Type: CardinalNumber</div>

```
c := 2**a
```

$$Aleph\,(1)$$

<div align="right">Type: CardinalNumber</div>

```
f := 2**c
```

$$Aleph\,(2)$$

<div align="right">Type: CardinalNumber</div>

9.7   CartesianTensor

`CartesianTensor(i0,dim,R)` provides Cartesian tensors with components belonging to a commutative ring `R`. Tensors can be described as a generalization of vectors and matrices. This gives a concise *tensor algebra* for multilinear objects supported by the `CartesianTensor` domain. You can form the inner or outer product of any two tensors and you can add or subtract tensors with the same number of components. Additionally, various forms of traces and transpositions are useful.

The `CartesianTensor` constructor allows you to specify the minimum index for subscripting. In what follows we discuss in detail how to manipulate tensors.

Here we construct the domain of Cartesian tensors of dimension 2 over the integers, with indices starting at 1.

```
CT := CARTEN(i0 := 1, 2, Integer)
```

$$CartesianTensor(1, 2, Integer)$$

<div align="right">Type: Domain</div>

Forming tensors

Scalars can be converted to tensors of rank zero.

```
t0:  CT := 8
```

$$8$$

<div align="center">Type: CartesianTensor(1,2,Integer)</div>

```
rank t0
```

$$0$$

<div align="right">Type: NonNegativeInteger</div>

Vectors (mathematical direct products, rather than one dimensional array structures) can be converted to tensors of rank one.

```
v:  DirectProduct(2, Integer) := directProduct [3,4]
```

$$[3, 4]$$

<div align="center">Type: DirectProduct(2,Integer)</div>

```
Tv:  CT := v
```

$$[3, 4]$$

<div align="center">Type: CartesianTensor(1,2,Integer)</div>

Matrices can be converted to tensors of rank two.

```
m:  SquareMatrix(2, Integer) := matrix [ [1,2],[4,5] ]
```

$$\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix}$$

                                    Type: SquareMatrix(2,Integer)

Tm: CT := m

$$\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix}$$

                                 Type: CartesianTensor(1,2,Integer)

n: SquareMatrix(2, Integer) := matrix [[2,3],[0,1]]

$$\begin{bmatrix} 2 & 3 \\ 0 & 1 \end{bmatrix}$$

                                    Type: SquareMatrix(2,Integer)

Tn: CT := n

$$\begin{bmatrix} 2 & 3 \\ 0 & 1 \end{bmatrix}$$

                                 Type: CartesianTensor(1,2,Integer)

In general, a tensor of rank k can be formed by making a list of rank k-1 tensors
or, alternatively, a k-deep nested list of lists.

t1: CT := [2, 3]

$$[2, 3]$$

                                 Type: CartesianTensor(1,2,Integer)

rank t1

$$1$$

                                    Type: PositiveInteger

t2: CT := [t1, t1]

$$\begin{bmatrix} 2 & 3 \\ 2 & 3 \end{bmatrix}$$

                    Type: CartesianTensor(1,2,Integer)

t3: CT := [t2, t2]

$$\begin{bmatrix} \begin{bmatrix} 2 & 3 \\ 2 & 3 \end{bmatrix}, & \begin{bmatrix} 2 & 3 \\ 2 & 3 \end{bmatrix} \end{bmatrix}$$

                    Type: CartesianTensor(1,2,Integer)

tt: CT := [t3, t3]; tt := [tt, tt]

$$\begin{bmatrix} \begin{bmatrix} \begin{bmatrix} 2 & 3 \\ 2 & 3 \\ 2 & 3 \\ 2 & 3 \end{bmatrix} & \begin{bmatrix} 2 & 3 \\ 2 & 3 \\ 2 & 3 \\ 2 & 3 \end{bmatrix} \end{bmatrix}, & \begin{bmatrix} \begin{bmatrix} 2 & 3 \\ 2 & 3 \\ 2 & 3 \\ 2 & 3 \end{bmatrix} & \begin{bmatrix} 2 & 3 \\ 2 & 3 \\ 2 & 3 \\ 2 & 3 \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

                    Type: CartesianTensor(1,2,Integer)

rank tt

                              5

                         Type: PositiveInteger

Multiplication

Given two tensors of rank `k1` and `k2`, the outer **product** forms a new tensor of rank `k1+k2`. Here

$$T_{mn}(i, j, k, l) = T_m(i, j) \; T_n(k, l)$$

Tmn := product(Tm, Tn)

$$\begin{bmatrix} \begin{bmatrix} 2 & 3 \\ 0 & 1 \end{bmatrix} & \begin{bmatrix} 4 & 6 \\ 0 & 2 \end{bmatrix} \\ \begin{bmatrix} 8 & 12 \\ 0 & 4 \end{bmatrix} & \begin{bmatrix} 10 & 15 \\ 0 & 5 \end{bmatrix} \end{bmatrix}$$

<div align="center">Type: CartesianTensor(1,2,Integer)</div>

The inner product (**contract**) forms a tensor of rank `k1+k2-2`. This product generalizes the vector dot product and matrix-vector product by summing component products along two indices.

Here we sum along the second index of T_m and the first index of T_v. Here

$$T_{mv} = \sum_{j=1}^{\dim} T_m(i,j)\ T_v(j)$$

```
Tmv := contract(Tm,2,Tv,1)
```

<div align="center">$[11, 32]$</div>

<div align="center">Type: CartesianTensor(1,2,Integer)</div>

The multiplication operator "`*`" is scalar multiplication or an inner product depending on the ranks of the arguments.

If either argument is rank zero it is treated as scalar multiplication. Otherwise, `a*b` is the inner product summing the last index of `a` with the first index of `b`.

```
Tm*Tv
```

<div align="center">$[11, 32]$</div>

<div align="center">Type: CartesianTensor(1,2,Integer)</div>

This definition is consistent with the inner product on matrices and vectors.

```
Tmv = m * v
```

<div align="center">$[11, 32] = [11, 32]$</div>

<div align="center">Type: Equation CartesianTensor(1,2,Integer)</div>

Selecting Components

For tensors of low rank (that is, four or less), components can be selected by applying the tensor to its indices.

```
t0()
```

                                        8

                                                Type: PositiveInteger

t1(1+1)

                                        3

                                                Type: PositiveInteger

t2(2,1)

                                        2

                                                Type: PositiveInteger

t3(2,1,2)

                                        3

                                                Type: PositiveInteger

Tmn(2,1,2,1)

                                        0

                                                Type: NonNegativeInteger

A general indexing mechanism is provided for a list of indices.

t0[]

                                        8

                                                Type: PositiveInteger

t1[2]

                                        3

t2[2,1]

$$2$$

Type: PositiveInteger

The general mechanism works for tensors of arbitrary rank, but is somewhat less efficient since the intermediate index list must be created.

t3[2,1,2]

$$3$$

Type: PositiveInteger

Tmn[2,1,2,1]

$$0$$

Type: NonNegativeInteger

Contraction

A "contraction" between two tensors is an inner product, as we have seen above. You can also contract a pair of indices of a single tensor. This corresponds to a "trace" in linear algebra. The expression `contract(t,k1,k2)` forms a new tensor by summing the diagonal given by indices in position `k1` and `k2`.

This is the tensor given by

$$xT_{mn} = \sum_{k=1}^{\dim} T_{mn}(k, k, i, j)$$

cTmn := contract(Tmn,1,2)

$$\begin{bmatrix} 12 & 18 \\ 0 & 6 \end{bmatrix}$$

Type: CartesianTensor(1,2,Integer)

Since `Tmn` is the outer product of matrix `m` and matrix `n`, the above is equivalent to this.

`trace(m) * n`

$$\begin{bmatrix} 12 & 18 \\ 0 & 6 \end{bmatrix}$$

                              Type: SquareMatrix(2,Integer)

In this and the next few examples, we show all possible contractions of `Tmn` and their matrix algebra equivalents.

`contract(Tmn,1,2) = trace(m) * n`

$$\begin{bmatrix} 12 & 18 \\ 0 & 6 \end{bmatrix} = \begin{bmatrix} 12 & 18 \\ 0 & 6 \end{bmatrix}$$

                          Type: Equation CartesianTensor(1,2,Integer)

`contract(Tmn,1,3) = transpose(m) * n`

$$\begin{bmatrix} 2 & 7 \\ 4 & 11 \end{bmatrix} = \begin{bmatrix} 2 & 7 \\ 4 & 11 \end{bmatrix}$$

                          Type: Equation CartesianTensor(1,2,Integer)

`contract(Tmn,1,4) = transpose(m) * transpose(n)`

$$\begin{bmatrix} 14 & 4 \\ 19 & 5 \end{bmatrix} = \begin{bmatrix} 14 & 4 \\ 19 & 5 \end{bmatrix}$$

                          Type: Equation CartesianTensor(1,2,Integer)

`contract(Tmn,2,3) = m * n`

$$\begin{bmatrix} 2 & 5 \\ 8 & 17 \end{bmatrix} = \begin{bmatrix} 2 & 5 \\ 8 & 17 \end{bmatrix}$$

                          Type: Equation CartesianTensor(1,2,Integer)

```
contract(Tmn,2,4) = m * transpose(n)
```

$$\begin{bmatrix} 8 & 2 \\ 23 & 5 \end{bmatrix} = \begin{bmatrix} 8 & 2 \\ 23 & 5 \end{bmatrix}$$

<div align="center">Type: Equation CartesianTensor(1,2,Integer)</div>

```
contract(Tmn,3,4) = trace(n) * m
```

$$\begin{bmatrix} 3 & 6 \\ 12 & 15 \end{bmatrix} = \begin{bmatrix} 3 & 6 \\ 12 & 15 \end{bmatrix}$$

<div align="center">Type: Equation CartesianTensor(1,2,Integer)</div>

Transpositions

You can exchange any desired pair of indices using the **transpose** operation.

Here the indices in positions one and three are exchanged, that is, $tT_{mn}(i,j,k,l) = T_{mn}(k,j,i,l)$.

```
tTmn := transpose(Tmn,1,3)
```

$$\begin{bmatrix} \begin{bmatrix} 2 & 3 \\ 8 & 12 \end{bmatrix} & \begin{bmatrix} 4 & 6 \\ 10 & 15 \end{bmatrix} \\ \begin{bmatrix} 0 & 1 \\ 0 & 4 \end{bmatrix} & \begin{bmatrix} 0 & 2 \\ 0 & 5 \end{bmatrix} \end{bmatrix}$$

<div align="center">Type: CartesianTensor(1,2,Integer)</div>

If no indices are specified, the first and last index are exchanged.

```
transpose Tmn
```

$$\begin{bmatrix} \begin{bmatrix} 2 & 8 \\ 0 & 0 \\ 3 & 12 \\ 1 & 4 \end{bmatrix} & \begin{bmatrix} 4 & 10 \\ 0 & 0 \\ 6 & 15 \\ 2 & 5 \end{bmatrix} \end{bmatrix}$$

<div align="center">Type: CartesianTensor(1,2,Integer)</div>

This is consistent with the matrix transpose.

```
transpose Tm = transpose m
```

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 2 & 5 \end{bmatrix}$$

<div align="center">Type: Equation CartesianTensor(1,2,Integer)</div>

If a more complicated reordering of the indices is required, then the **reindex** operation can be used. This operation allows the indices to be arbitrarily permuted.

This defines $rT_{mn}(i, j, k, l) = T_{mn}(i, l, j, k)$.

```
rTmn := reindex(Tmn, [1,4,2,3])
```

$$\begin{bmatrix} \begin{bmatrix} 2 & 0 \\ 4 & 0 \\ 8 & 0 \\ 10 & 0 \end{bmatrix} & \begin{bmatrix} 3 & 1 \\ 6 & 2 \\ 12 & 4 \\ 15 & 5 \end{bmatrix} \end{bmatrix}$$

<div align="center">Type: CartesianTensor(1,2,Integer)</div>

Arithmetic

Tensors of equal rank can be added or subtracted so arithmetic expressions can be used to produce new tensors.

```
tt := transpose(Tm)*Tn - Tn*transpose(Tm)
```

$$\begin{bmatrix} -6 & -16 \\ 2 & 6 \end{bmatrix}$$

<div align="center">Type: CartesianTensor(1,2,Integer)</div>

```
Tv*(tt+Tn)
```

$$[-4, -11]$$

<div align="center">Type: CartesianTensor(1,2,Integer)</div>

```
reindex(product(Tn,Tn),[4,3,2,1])+3*Tn*product(Tm,Tm)
```

$$
\begin{bmatrix}
\begin{bmatrix} 46 & 84 \\ 174 & 212 \end{bmatrix} & \begin{bmatrix} 57 & 114 \\ 228 & 285 \end{bmatrix} \\
\begin{bmatrix} 18 & 24 \\ 57 & 63 \end{bmatrix} & \begin{bmatrix} 17 & 30 \\ 63 & 76 \end{bmatrix}
\end{bmatrix}
$$

<div align="right">Type: CartesianTensor(1,2,Integer)</div>

Specific Tensors

Two specific tensors have properties which depend only on the dimension.
The Kronecker delta satisfies

```
            +-
            |   1  if i  = j
delta(i,j) = |
            |   0  if i ^= j
            +-
```

```
delta:  CT := kroneckerDelta()
```

$$
\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}
$$

<div align="right">Type: CartesianTensor(1,2,Integer)</div>

This can be used to reindex via contraction.

```
contract(Tmn, 2, delta, 1) = reindex(Tmn, [1,3,4,2])
```

$$
\begin{bmatrix}
\begin{bmatrix} 2 & 4 \\ 3 & 6 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 1 & 2 \end{bmatrix} \\
\begin{bmatrix} 8 & 10 \\ 12 & 15 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 4 & 5 \end{bmatrix}
\end{bmatrix}
=
\begin{bmatrix}
\begin{bmatrix} 2 & 4 \\ 3 & 6 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 1 & 2 \end{bmatrix} \\
\begin{bmatrix} 8 & 10 \\ 12 & 15 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 4 & 5 \end{bmatrix}
\end{bmatrix}
$$

<div align="right">Type: Equation CartesianTensor(1,2,Integer)</div>

The Levi Civita symbol determines the sign of a permutation of indices.

```
epsilon:CT := leviCivitaSymbol()
```

$$
\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}
$$

```
                              Type:  CartesianTensor(1,2,Integer)
```

Here we have:

```
epsilon(i1,...,idim)
    = +1  if i1,...,idim is an even permutation of i0,...,i0+dim-1
    = -1  if i1,...,idim is an  odd permutation of i0,...,i0+dim-1
    =  0  if i1,...,idim is not   a permutation of i0,...,i0+dim-1
```

This property can be used to form determinants.

```
contract(epsilon*Tm*epsilon, 1,2) = 2 * determinant m
```

$$-6 = -6$$

```
            Type:  Equation CartesianTensor(1,2,Integer)
```

Properties of the CartesianTensor domain

`GradedModule(R,E)` denotes "E-graded R-module", that is, a collection of `R`-modules indexed by an abelian monoid `E`. An element `g` of `G[s]` for some specific `s` in `E` is said to be an element of `G` with **degree s**. Sums are defined in each module `G[s]` so two elements of `G` can be added if they have the same degree. Morphisms can be defined and composed by degree to give the mathematical category of graded modules.

`GradedAlgebra(R,E)` denotes "E-graded R-algebra." A graded algebra is a graded module together with a degree preserving `R`-bilinear map, called the **product**.

```
degree(product(a,b))    = degree(a) + degree(b)

product(r*a,b)          = product(a,r*b) = r*product(a,b)
product(a1+a2,b)        = product(a1,b) + product(a2,b)
product(a,b1+b2)        = product(a,b1) + product(a,b2)
product(a,product(b,c)) = product(product(a,b),c)
```

The domain `CartesianTensor(i0, dim, R)` belongs to the category `GradedAlgebra(R, NonNegativeInteger)`. The non-negative integer **degree** is the tensor rank and the graded algebra **product** is the tensor outer product. The graded module addition captures the notion that only tensors of equal rank can be added.

If `V` is a vector space of dimension `dim` over `R`, then the tensor module `T[k](V)` is defined as

```
T[0](V) = R
T[k](V) = T[k-1](V) * V
```

where `*` denotes the R-module tensor **product**. `CartesianTensor(i0,dim,R)` is the graded algebra in which the degree `k` module is `T[k](V)`.

Tensor Calculus

It should be noted here that often tensors are used in the context of tensor-valued manifold maps. This leads to the notion of covariant and contravariant bases with tensor component functions transforming in specific ways under a change of coordinates on the manifold. This is no more directly supported by the `CartesianTensor` domain than it is by the `Vector` domain. However, it is possible to have the components implicitly represent component maps by choosing a polynomial or expression type for the components. In this case, it is up to the user to satisfy any constraints which arise on the basis of this interpretation.

9.8   Character

The members of the domain `Character` are values representing letters, numerals and other text elements. For more information on related topics, see 9.9 on page 491 and 9.77 on page 804.

Characters can be obtained using `String` notation.

```
chars := [char "a", char "A", char "X", char "8", char "+"]
```

$$[a, A, X, 8, +]$$

                                                    Type: List Character

Certain characters are available by name. This is the blank character.

```
space()
```

                                                    Type: Character

This is the quote that is used in strings.

```
quote()
```

$$"$$

<div align="right">Type: Character</div>

This is the escape character that allows quotes and other characters within strings.

```
escape()
```

$$-$$

<div align="right">Type: Character</div>

Characters are represented as integers in a machine-dependent way. The integer value can be obtained using the **ord** operation. It is always true that `char(ord c) = c` and `ord(char i) = i`, provided that `i` is in the range `0..size()$Character-1`.

```
[ord c for c in chars]
```

$$[97, 65, 88, 56, 43]$$

<div align="right">Type: List Integer</div>

The **lowerCase** operation converts an upper case letter to the corresponding lower case letter. If the argument is not an upper case letter, then it is returned unchanged.

```
[upperCase c for c in chars]
```

$$[A, A, X, 8, +]$$

<div align="right">Type: List Character</div>

Likewise, the **upperCase** operation converts lower case letters to upper case.

```
[lowerCase c for c in chars]
```

$$[a, a, x, 8, +]$$

<div align="right">Type: List Character</div>

A number of tests are available to determine whether characters belong to certain families.

```
[alphabetic?  c for c in chars]
```

$$[\mathrm{true}, \mathrm{true}, \mathrm{true}, \mathrm{false}, \mathrm{false}]$$

                                        Type: List Boolean

```
[upperCase?  c for c in chars]
```

$$[\mathrm{false}, \mathrm{true}, \mathrm{true}, \mathrm{false}, \mathrm{false}]$$

                                        Type: List Boolean

```
[lowerCase?  c for c in chars]
```

$$[\mathrm{true}, \mathrm{false}, \mathrm{false}, \mathrm{false}, \mathrm{false}]$$

                                        Type: List Boolean

```
[digit?  c for c in chars]
```

$$[\mathrm{false}, \mathrm{false}, \mathrm{false}, \mathrm{true}, \mathrm{false}]$$

                                        Type: List Boolean

```
[hexDigit?  c for c in chars]
```

$$[\mathrm{true}, \mathrm{true}, \mathrm{false}, \mathrm{true}, \mathrm{false}]$$

                                        Type: List Boolean

```
[alphanumeric?  c for c in chars]
```

$$[\mathrm{true}, \mathrm{true}, \mathrm{true}, \mathrm{true}, \mathrm{false}]$$

                                        Type: List Boolean

9.9 CharacterClass

The `CharacterClass` domain allows classes of characters to be defined and manipulated efficiently.

Character classes can be created by giving either a string or a list of characters.

```
cl1 := charClass [char "a", char "e", char "i", char "o", char
"u", char "y"]
```

$$\text{"aeiouy"}$$

                                        Type: CharacterClass

```
cl2 := charClass "bcdfghjklmnpqrstvwxyz"
```

$$\text{"bcdfghjklmnpqrstvwxyz"}$$

                                        Type: CharacterClass

A number of character classes are predefined for convenience.

```
digit()
```

$$\text{"0123456789"}$$

                                        Type: CharacterClass

```
hexDigit()
```

$$\text{"0123456789ABCDEFabcdef"}$$

                                        Type: CharacterClass

```
upperCase()
```

$$\text{"ABCDEFGHIJKLMNOPQRSTUVWXYZ"}$$

                                        Type: CharacterClass

```
lowerCase()
```

```
                      "abcdefghijklmnopqrstuvwxyz"
```

```
                                    Type:   CharacterClass
```

```
alphabetic()
```

```
      "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
```

```
                                    Type:   CharacterClass
```

```
alphanumeric()
```

```
 "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
```

```
                                    Type:   CharacterClass
```

You can quickly test whether a character belongs to a class.

```
member?(char "a", cl1)
```

```
                              true
```

```
                                       Type:   Boolean
```

```
member?(char "a", cl2)
```

```
                              false
```

```
                                       Type:   Boolean
```

Classes have the usual set operations because the `CharacterClass` domain belongs to the category `FiniteSetAggregate(Character)`.

```
intersect(cl1, cl2)
```

```
                              "y"
```

```
                                    Type:   CharacterClass
```

```
union(cl1,cl2)
```

```
                         "abcdefghijklmnopqrstuvwxyz"
```

<div align="right">Type: CharacterClass</div>

```
difference(cl1,cl2)
```

```
                                  "aeiou"
```

<div align="right">Type: CharacterClass</div>

```
intersect(complement(cl1),cl2)
```

```
                         "bcdfghjklmnpqrstvwxz"
```

<div align="right">Type: CharacterClass</div>

You can modify character classes by adding or removing characters.

```
insert!(char "a", cl2)
```

```
                         "abcdfghjklmnpqrstvwxyz"
```

<div align="right">Type: CharacterClass</div>

```
remove!(char "b", cl2)
```

```
                         "acdfghjklmnpqrstvwxyz"
```

<div align="right">Type: CharacterClass</div>

For more information on related topics, see 9.8 on page 488 and 9.77 on page 804.

9.10   CliffordAlgebra

`CliffordAlgebra(n,K,Q)` defines a vector space of dimension 2^n over the field K with a given quadratic form `Q`. If $\{e_1, \ldots, e_n\}$ is a basis for K^n then

```
{ 1,
  e(i) 1 <= i <= n,
  e(i1)*e(i2) 1 <= i1 < i2 <=n,
  ...,
  e(1)*e(2)*...*e(n) }
```

is a basis for the Clifford algebra. The algebra is defined by the relations

```
e(i)*e(i) = Q(e(i))
e(i)*e(j) = -e(j)*e(i),  i ^= j
```

Examples of Clifford Algebras are gaussians (complex numbers), quaternions, exterior algebras and spin algebras.

9.10.1   The Complex Numbers as a Clifford Algebra

This is the field over which we will work, rational functions with integer coefficients.

```
K := Fraction Polynomial Integer
```

$$\text{Fraction Polynomial Integer}$$

                                                    Type: Domain

We use this matrix for the quadratic form.

```
m := matrix [ [-1] ]
```

$$\begin{bmatrix} -1 \end{bmatrix}$$

                                                    Type: Matrix Integer

We get complex arithmetic by using this domain.

```
C := CliffordAlgebra(1, K, quadraticForm m)
```

$$\text{CliffordAlgebra(1,Fraction Polynomial Integer,MATRIX)}$$

                                                    Type: Domain

Here is `i`, the usual square root of `-1`.

```
i:  C := e(1)
```

$$e_1$$

        Type: CliffordAlgebra(1,Fraction Polynomial Integer,MATRIX)

Here are some examples of the arithmetic.

```
x := a + b * i
```

$$a + b \ e_1$$

    Type: CliffordAlgebra(1,Fraction Polynomial Integer,MATRIX)

```
y := c + d * i
```

$$c + d \ e_1$$

    Type: CliffordAlgebra(1,Fraction Polynomial Integer,MATRIX)

See for examples of Axiom's constructor implementing complex numbers.

```
x * y
```

$$-b \ d + a \ c + (a \ d + b \ c) \ e_1$$

    Type: CliffordAlgebra(1,Fraction Polynomial Integer,MATRIX)

9.10.2   The Quaternion Numbers as a Clifford Algebra

This is the field over which we will work, rational functions with integer coefficients.

```
K := Fraction Polynomial Integer
```

$$\text{Fraction Polynomial Integer}$$

                                    Type: Domain

We use this matrix for the quadratic form.

```
m := matrix [ [-1,0],[0,-1] ]
```

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$$

<div align="right">Type: Matrix Integer</div>

The resulting domain is the quaternions.

```
H := CliffordAlgebra(2, K, quadraticForm m)
```

<div align="center">CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)</div>

<div align="right">Type: Domain</div>

We use Hamilton's notation for i,j,k.

```
i:  H := e(1)
```

$$e_1$$

<div align="center">Type: CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)</div>

```
j:  H := e(2)
```

$$e_2$$

<div align="center">Type: CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)</div>

```
k:  H := i * j
```

$$e_1\ e_2$$

<div align="center">Type: CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)</div>

```
x := a + b * i + c * j + d * k
```

$$a + b\ e_1 + c\ e_2 + d\ e_1\ e_2$$

<div align="center">Type: CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)</div>

```
y := e + f * i + g * j + h * k
```

$$e + f\ e_1 + g\ e_2 + h\ e_1\ e_2$$

```
    Type:  CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)
```

```
x + y
```

$$e + a + (f + b) \ e_1 + (g + c) \ e_2 + (h + d) \ e_1 \ e_2$$

```
    Type:  CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)
```

```
x * y
```

$$-d \ h - c \ g - b \ f + a \ e + (c \ h - d \ g + a \ f + b \ e) \ e_1 +$$

$$(-b \ h + a \ g + d \ f + c \ e) \ e_2 + (a \ h + b \ g - c \ f + d \ e) \ e_1 \ e_2$$

```
    Type:  CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)
```

See 9.64 on page 745 for examples of Axiom's constructor implementing quaternions.

```
y * x
```

$$-d \ h - c \ g - b \ f + a \ e + (-c \ h + d \ g + a \ f + b \ e) \ e_1 +$$

$$(b \ h + a \ g - d \ f + c \ e) \ e_2 + (a \ h - b \ g + c \ f + d \ e) \ e_1 \ e_2$$

```
    Type:  CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)
```

9.10.3   The Exterior Algebra on a Three Space

This is the field over which we will work, rational functions with integer coefficients.

```
K := Fraction Polynomial Integer
```

Fraction Polynomial Integer

```
                                             Type:  Domain
```

If we chose the three by three zero quadratic form, we obtain the exterior algebra on e(1),e(2),e(3).

```
Ext := CliffordAlgebra(3, K, quadraticForm 0)
```

CliffordAlgebra(3,Fraction Polynomial Integer,MATRIX)

                                                    Type: Domain

This is a three dimensional vector algebra. We define `i`, `j`, `k` as the unit vectors.

```
i:  Ext := e(1)
```

$$e_1$$

        Type: CliffordAlgebra(3,Fraction Polynomial Integer,MATRIX)

```
j:  Ext := e(2)
```

$$e_2$$

        Type: CliffordAlgebra(3,Fraction Polynomial Integer,MATRIX)

```
k:  Ext := e(3)
```

$$e_3$$

        Type: CliffordAlgebra(3,Fraction Polynomial Integer,MATRIX)

Now it is possible to do arithmetic.

```
x := x1*i + x2*j + x3*k
```

$$x1 \ e_1 + x2 \ e_2 + x3 \ e_3$$

        Type: CliffordAlgebra(3,Fraction Polynomial Integer,MATRIX)

```
y := y1*i + y2*j + y3*k
```

$$y1 \ e_1 + y2 \ e_2 + y3 \ e_3$$

        Type: CliffordAlgebra(3,Fraction Polynomial Integer,MATRIX)

```
x + y
```

$$(y1 + x1)\ e_1 + (y2 + x2)\ e_2 + (y3 + x3)\ e_3$$

    Type: CliffordAlgebra(3,Fraction Polynomial Integer,MATRIX)

```
x * y + y * x
```

$$0$$

    Type: CliffordAlgebra(3,Fraction Polynomial Integer,MATRIX)

On an **n** space, a grade **p** form has a dual **n-p** form. In particular, in three space the dual of a grade two element identifies e1*e2->e3, e2*e3->e1, e3*e1->e2.

```
dual2 a == coefficient(a,[2,3]) * i + coefficient(a,[3,1]) * j +
coefficient(a,[1,2]) * k
```

                                                            Type: Void

The vector cross product is then given by this.

```
dual2(x*y)
```

   Compiling function dual2 with type CliffordAlgebra(3,Fraction
      Polynomial Integer,MATRIX) -> CliffordAlgebra(3,Fraction
      Polynomial Integer,MATRIX)

$$(x2\ y3 - x3\ y2)\ e_1 + (-x1\ y3 + x3\ y1)\ e_2 + (x1\ y2 - x2\ y1)\ e_3$$

    Type: CliffordAlgebra(3,Fraction Polynomial Integer,MATRIX)

9.10.4   The Dirac Spin Algebra

In this section we will work over the field of rational numbers.

```
K := Fraction Integer
```

$$\text{Fraction Integer}$$

We define the quadratic form to be the Minkowski space-time metric.

```
g := matrix [ [1,0,0,0], [0,-1,0,0], [0,0,-1,0], [0,0,0,-1] ]
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

Type: Matrix Integer

We obtain the Dirac spin algebra used in Relativistic Quantum Field Theory.

```
D := CliffordAlgebra(4,K, quadraticForm g)
```

CliffordAlgebra(4,Fraction Integer,MATRIX)

Type: Domain

The usual notation for the basis is γ with a superscript. For Axiom input we
will use gam(i):

```
gam := [e(i)$D for i in 1..4]
```

$$[e_1, e_2, e_3, e_4]$$

Type: List CliffordAlgebra(4,Fraction Integer,MATRIX)

There are various contraction identities of the form

```
g(l,t)*gam(l)*gam(m)*gam(n)*gam(r)*gam(s)*gam(t) =
     2*(gam(s)gam(m)gam(n)gam(r) + gam(r)*gam(n)*gam(m)*gam(s))
```

where a sum over l and t is implied.

Verify this identity for particular values of m,n,r,s.

```
m := 1; n:= 2; r := 3; s := 4;
```

Type: PositiveInteger

```
lhs := reduce(+, [reduce(+, [
g(l,t)*gam(l)*gam(m)*gam(n)*gam(r)*gam(s)*gam(t) for l in 1..4])
for t in 1..4])
```

$$-4 \, e_1 \, e_2 \, e_3 \, e_4$$

Type: CliffordAlgebra(4,Fraction Integer,MATRIX)

```
rhs := 2*(gam s * gam m*gam n*gam r + gam r*gam n*gam m*gam s)
```

$$-4 \, e_1 \, e_2 \, e_3 \, e_4$$

Type: CliffordAlgebra(4,Fraction Integer,MATRIX)

9.11 Complex

The `Complex` constructor implements complex objects over a commutative ring R. Typically, the ring R is `Integer`, `Fraction Integer`, `Float` or `DoubleFloat`. R can also be a symbolic type, like `Polynomial Integer`. For more information about the numerical and graphical aspects of complex numbers, see 8.1 on page 345.

Complex objects are created by the **complex** operation.

```
a := complex(4/3,5/2)
```

$$\frac{4}{3} + \frac{5}{2} \, i$$

Type: Complex Fraction Integer

```
b := complex(4/3,-5/2)
```

$$\frac{4}{3} - \frac{5}{2} \, i$$

Type: Complex Fraction Integer

The standard arithmetic operations are available.

```
a + b
```

$$\frac{8}{3}$$

<div align="right">Type: Complex Fraction Integer</div>

a - b

$$5\ i$$

<div align="right">Type: Complex Fraction Integer</div>

a * b

$$\frac{289}{36}$$

<div align="right">Type: Complex Fraction Integer</div>

If R is a field, you can also divide the complex objects.

a / b

$$-\frac{161}{289}+\frac{240}{289}\ i$$

<div align="right">Type: Complex Fraction Integer</div>

Use a conversion (**??** on page **??** in Section **??** on page **??**) to view the last object as a fraction of complex integers.

% :: Fraction Complex Integer

$$\frac{-15+8\ i}{15+8\ i}$$

<div align="right">Type: Fraction Complex Integer</div>

The predefined macro %i is defined to be complex(0,1).

3.4 + 6.7 * %i

$$3.4+6.7\ i$$

<div align="right">Type: Complex Float</div>

You can also compute the **conjugate** and **norm** of a complex number.

`conjugate a`

$$\frac{4}{3} - \frac{5}{2}\, i$$

<div align="right">Type: Complex Fraction Integer</div>

`norm a`

$$\frac{289}{36}$$

<div align="right">Type: Fraction Integer</div>

The **real** and **imag** operations are provided to extract the real and imaginary parts, respectively.

`real a`

$$\frac{4}{3}$$

<div align="right">Type: Fraction Integer</div>

`imag a`

$$\frac{5}{2}$$

<div align="right">Type: Fraction Integer</div>

The domain `Complex Integer` is also called the Gaussian integers. If `R` is the integers (or, more generally, a `EuclideanDomain`), you can compute greatest common divisors.

`gcd(13 - 13*%i,31 + 27*%i)`

$$5 + i$$

<div align="right">Type: Complex Integer</div>

You can also compute least common multiples.

```
lcm(13 - 13*%i,31 + 27*%i)
```

$$143 - 39 \; i$$

<div align="right">Type: Complex Integer</div>

You can **factor** Gaussian integers.

```
factor(13 - 13*%i)
```

$$-(1 + i) \; (2 + 3 \; i) \; (3 + 2 \; i)$$

<div align="right">Type: Factored Complex Integer</div>

```
factor complex(2,0)
```

$$-i \; (1 + i)^2$$

<div align="right">Type: Factored Complex Integer</div>

9.12   ContinuedFraction

Continued fractions have been a fascinating and useful tool in mathematics for well over three hundred years. Axiom implements continued fractions for fractions of any Euclidean domain. In practice, this usually means rational numbers. In this section we demonstrate some of the operations available for manipulating both finite and infinite continued fractions. It may be helpful if you review 9.76 on page 801 to remind yourself of some of the operations with streams.

The `ContinuedFraction` domain is a field and therefore you can add, subtract, multiply and divide the fractions.

The **continuedFraction** operation converts its fractional argument to a continued fraction.

```
c := continuedFraction(314159/100000)
```

$$3 + \frac{1|}{|7} + \frac{1|}{|15} + \frac{1|}{|1} + \frac{1|}{|25} + \frac{1|}{|1} + \frac{1|}{|7} + \frac{1|}{|4}$$

<div align="right">Type: ContinuedFraction Integer</div>

This display is a compact form of the bulkier

```
    3 +                     1
          --------------------------------
        7 +                   1
              --------------------------
            15 +                1
                  ----------------------
                1 +             1
                      ------------------
                    25 +        1
                          -------------
                        1 +     1
                              ---------
                            7 +   1
                                  -----
                                    4
```

You can write any rational number in a similar form. The fraction will be finite and you can always take the "numerators" to be 1. That is, any rational number can be written as a simple, finite continued fraction of the form

```
    a(1) +              1
            -------------------------
          a(2) +            1
                -------------------
                a(3) +
                      .
                        .
                          .
                                1
                          -------------
                        a(n-1) +  1
                                 ----
                                a(n)
```

The a_i are called partial quotients and the operation **partialQuotients** creates a stream of them.

partialQuotients c

$$[3, 7, 15, 1, 25, 1, 7, \ldots]$$

Type: Stream Integer

By considering more and more of the fraction, you get the **convergents**. For example, the first convergent is a_1, the second is $a_1 + 1/a_2$ and so on.

`convergents c`

$$\left[3, \frac{22}{7}, \frac{333}{106}, \frac{355}{113}, \frac{9208}{2931}, \frac{9563}{3044}, \frac{76149}{24239}, \ldots\right]$$

<div align="right">Type: Stream Fraction Integer</div>

Since this is a finite continued fraction, the last convergent is the original rational number, in reduced form. The result of **approximants** is always an infinite stream, though it may just repeat the "last" value.

`approximants c`

$$\left[3, \frac{22}{7}, \frac{333}{106}, \frac{355}{113}, \frac{9208}{2931}, \frac{9563}{3044}, \frac{76149}{24239}, \ldots\right]$$

<div align="right">Type: Stream Fraction Integer</div>

Inverting `c` only changes the partial quotients of its fraction by inserting a `0` at the beginning of the list.

`pq := partialQuotients(1/c)`

$$[0, 3, 7, 15, 1, 25, 1, \ldots]$$

<div align="right">Type: Stream Integer</div>

Do this to recover the original continued fraction from this list of partial quotients. The three-argument form of the **continuedFraction** operation takes an element which is the whole part of the fraction, a stream of elements which are the numerators of the fraction, and a stream of elements which are the denominators of the fraction.

`continuedFraction(first pq,repeating [1],rest pq)`

$$\frac{1|}{|3} + \frac{1|}{|7} + \frac{1|}{|15} + \frac{1|}{|1} + \frac{1|}{|25} + \frac{1|}{|1} + \frac{1|}{|7} + \ldots$$

<div align="right">Type: ContinuedFraction Integer</div>

The streams need not be finite for **continuedFraction**. Can you guess which irrational number has the following continued fraction? See the end of this section for the answer.

```
z:=continuedFraction(3,repeating [1],repeating [3,6])
```

$$3 + \frac{1|}{|3} + \frac{1|}{|6} + \frac{1|}{|3} + \frac{1|}{|6} + \frac{1|}{|3} + \frac{1|}{|6} + \frac{1|}{|3} + \dots$$

<div align="right">Type: ContinuedFraction Integer</div>

In 1737 Euler discovered the infinite continued fraction expansion

```
  e - 1                   1
  ----- = ----------------------
    2       1 +            1
                   ----------------
                     6 +       1
                         ------------
                          10 +    1
                               --------
                                14 + ...
```

We use this expansion to compute rational and floating point approximations of e.[2]

By looking at the above expansion, we see that the whole part is `0` and the numerators are all equal to `1`. This constructs the stream of denominators.

```
dens:Stream Integer := cons(1,generate((x+->x+4),6))
```

$$[1, 6, 10, 14, 18, 22, 26, \dots]$$

<div align="right">Type: Stream Integer</div>

Therefore this is the continued fraction expansion for $(e - 1)/2$.

```
cf := continuedFraction(0,repeating [1],dens)
```

$$\frac{1|}{|1} + \frac{1|}{|6} + \frac{1|}{|10} + \frac{1|}{|14} + \frac{1|}{|18} + \frac{1|}{|22} + \frac{1|}{|26} + \dots$$

<div align="right">Type: ContinuedFraction Integer</div>

---

[2]For this and other interesting expansions, see C. D. Olds, *Continued Fractions,* New Mathematical Library, (New York: Random House, 1963), pp. 134–139.

These are the rational number convergents.

```
ccf := convergents cf
```

$$\left[0, 1, \frac{6}{7}, \frac{61}{71}, \frac{860}{1001}, \frac{15541}{18089}, \frac{342762}{398959}, \ldots\right]$$

<div align="right">Type: Stream Fraction Integer</div>

You can get rational convergents for e by multiplying by 2 and adding 1.

```
eConvergents := [2*e + 1 for e in ccf]
```

$$\left[1, 3, \frac{19}{7}, \frac{193}{71}, \frac{2721}{1001}, \frac{49171}{18089}, \frac{1084483}{398959}, \ldots\right]$$

<div align="right">Type: Stream Fraction Integer</div>

You can also compute the floating point approximations to these convergents.

```
eConvergents ::  Stream Float
```

$$[1.0, 3.0, 2.7142857142857142857, 2.7183098591549295775,$$

$$2.7182817182817182817, 2.7182818287356957267,$$

$$2.7182818284\ 585634113, \ldots]$$

<div align="right">Type: Stream Float</div>

Compare this to the value of e computed by the **exp** operation in `Float`.

```
exp 1.0
```

<div align="center">2.7182818284 590452354</div>

<div align="right">Type: Float</div>

In about 1658, Lord Brouncker established the following expansion for $4/\pi$,

```
    1 +                 1
         ----------------------
         2 +              9
```

```
                  -------------------
     2  +              25
                  ---------------
          2  +           49
                      -----------
               2  +       81
                          -------
                    2  + ...
```

Let's use this expansion to compute rational and floating point approximations for π.

```
cf := continuedFraction(1,[(2*i+1)**2 for i in 0..],repeating
[2])
```

$$1 + \frac{1|}{|2} + \frac{9|}{|2} + \frac{25|}{|2} + \frac{49|}{|2} + \frac{81|}{|2} + \frac{121|}{|2} + \frac{169|}{|2} + \ldots$$

> Type: ContinuedFraction Integer

```
ccf := convergents cf
```

$$\left[1, \frac{3}{2}, \frac{15}{13}, \frac{105}{76}, \frac{315}{263}, \frac{3465}{2578}, \frac{45045}{36979}, \ldots \right]$$

> Type: Stream Fraction Integer

```
piConvergents := [4/p for p in ccf]
```

$$\left[4, \frac{8}{3}, \frac{52}{15}, \frac{304}{105}, \frac{1052}{315}, \frac{10312}{3465}, \frac{147916}{45045}, \ldots \right]$$

> Type: Stream Fraction Integer

As you can see, the values are converging to $\pi = 3.14159265358979323846...$, but not very quickly.

```
piConvergents ::  Stream Float
```

$$[4.0, 2.6666666666\ 666666667, 3.4666666666\ 666666667,$$

$$2.8952380952\ 380952381, 3.3396825396\ 825396825,$$

$$2.9760461760\ 461760462, 3.2837384837\ 384837385, \ldots]$$

You need not restrict yourself to continued fractions of integers. Here is an expansion for a quotient of Gaussian integers.

```
continuedFraction((- 122 + 597*%i)/(4 - 4*%i))
```

$$-90 + 59\ i + \frac{1|}{|1 - 2\ i} + \frac{1|}{|-1 + 2\ i}$$

Type: ContinuedFraction Complex Integer

This is an expansion for a quotient of polynomials in one variable with rational number coefficients.

```
r :  Fraction UnivariatePolynomial(x,Fraction Integer)
```

Type: Void

```
r := ((x - 1) * (x - 2)) / ((x-3) * (x-4))
```

$$\frac{x^2 - 3\ x + 2}{x^2 - 7\ x + 12}$$

Type: Fraction UnivariatePolynomial(x,Fraction Integer)

```
continuedFraction r
```

$$1 + \frac{1|}{\left|\frac{1}{4}\ x - \frac{9}{8}\right.} + \frac{1|}{\left|\frac{16}{3}\ x - \frac{40}{3}\right.}$$

Type: ContinuedFraction UnivariatePolynomial(x,Fraction Integer)

To conclude this section, we give you evidence that

```
    z = 3 +             1
            ----------------------
            3 +           1
                -------------------
                6 +          1
                    ---------------
                    3 +       1
                        -----------
                        6 +    1
                            -------
                            3 + ...
```

is the expansion of $\sqrt{11}$.

```
[i*i for i in convergents(z) ::  Stream Float]
```

$$[9.0, 11.1111111111\ 11111111, 10.9944598337\ 9501385,$$

$$11.0002777777\ 77777778, 10.9999860763\ 98799786,$$

$$11.0000006979\ 29731039, 10.9999999650\ 15834446, \ldots]$$

<div align="right">Type: Stream Float</div>

9.13 CycleIndicators

This section is based upon the paper J. H. Redfield, "The Theory of Group-Reduced Distributions", American J. Math.,49 (1927) 433-455, and is an application of group theory to enumeration problems. It is a development of the work by P. A. MacMahon on the application of symmetric functions and Hammond operators to combinatorial theory.

The theory is based upon the power sum symmetric functions s_i which are the sum of the i-th powers of the variables. The cycle index of a permutation is an expression that specifies the sizes of the cycles of a permutation, and may be represented as a partition. A partition of a non-negative integer n is a collection of positive integers called its parts whose sum is n. For example, the partition $(3^2\ 2\ 1^2)$ will be used to represent $s_3^2 s_2 s_1^2$ and will indicate that the permutation has two cycles of length 3, one of length 2 and two of length 1. The cycle index of a permutation group is the sum of the cycle indices of its permutations divided by the number of permutations. The cycle indices of certain groups are provided.

The operation `complete` returns the cycle index of the symmetric group of order n for argument n. Alternatively, it is the n-th complete homogeneous symmetric function expressed in terms of power sum symmetric functions.

```
complete 1
```

<div align="center">(1)</div>

<div align="right">Type: SymmetricPolynomial Fraction Integer</div>

```
complete 2
```

$$\frac{1}{2}\ (2) + \frac{1}{2}\ (1^2)$$

<div align="center">Type: SymmetricPolynomial Fraction Integer</div>

complete 3

$$\frac{1}{3}\ (3) + \frac{1}{2}\ (2\ \ 1) + \frac{1}{6}\ (1^3)$$

<div align="center">Type: SymmetricPolynomial Fraction Integer</div>

complete 7

$$\frac{1}{7}\ (7) + \frac{1}{6}\ (6\ \ 1) + \frac{1}{10}\ (5\ \ 2) + \frac{1}{10}\ (5\ \ 1^2) + \frac{1}{12}\ (4\ \ 3) + \frac{1}{8}\ (4\ \ 2\ \ 1) +$$

$$\frac{1}{24}\ (4\ \ 1^3) + \frac{1}{18}\ (3^2\ 1) + \frac{1}{24}\ (3\ \ 2^2) + \frac{1}{12}\ (3\ \ 2\ \ 1^2) + \frac{1}{72}\ (3\ \ 1^4) +$$

$$\frac{1}{48}\ (2^3\ 1) + \frac{1}{48}\ (2^2\ 1^3) + \frac{1}{240}\ (2\ \ 1^5) + \frac{1}{5040}\ (1^7)$$

<div align="center">Type: SymmetricPolynomial Fraction Integer</div>

The operation `elementary` computes the n-th elementary symmetric function for argument `n`.

elementary 7

$$\frac{1}{7}\ (7) - \frac{1}{6}\ (6\ \ 1) - \frac{1}{10}\ (5\ \ 2) + \frac{1}{10}\ (5\ \ 1^2) - \frac{1}{12}\ (4\ \ 3) + \frac{1}{8}\ (4\ \ 2\ \ 1)$$

$$-\frac{1}{24}\ (4\ \ 1^3) + \frac{1}{18}\ (3^2\ 1) + \frac{1}{24}\ (3\ \ 2^2) - \frac{1}{12}\ (3\ \ 2\ \ 1^2) + \frac{1}{72}\ (3\ \ 1^4)$$

$$-\frac{1}{48}\ (2^3\ 1) + \frac{1}{48}\ (2^2\ 1^3) - \frac{1}{240}\ (2\ \ 1^5) + \frac{1}{5040}\ (1^7)$$

<div align="center">Type: SymmetricPolynomial Fraction Integer</div>

The operation `alternating` returns the cycle index of the alternating group having an even number of even parts in each cycle partition.

alternating 7

$$\tfrac{2}{7}\,(7) + \tfrac{1}{5}\,\left(5\ \ 1^2\right) + \tfrac{1}{4}\,\left(4\ \ 2\ \ 1\right) + \tfrac{1}{9}\,\left(3^2\ 1\right) + \tfrac{1}{12}\,\left(3\ \ 2^2\right) + \tfrac{1}{36}\,\left(3\ \ 1^4\right)+$$

$$\frac{1}{24}\,\left(2^2\ 1^3\right) + \frac{1}{2520}\,\left(1^7\right)$$

<div align="center">Type: SymmetricPolynomial Fraction Integer</div>

The operation `cyclic` returns the cycle index of the cyclic group.

`cyclic 7`

$$\frac{6}{7}\,(7) + \frac{1}{7}\,\left(1^7\right)$$

<div align="center">Type: SymmetricPolynomial Fraction Integer</div>

The operation `dihedral` is the cycle index of the dihedral group.

`dihedral 7`

$$\frac{3}{7}\,(7) + \frac{1}{2}\,\left(2^3\ 1\right) + \frac{1}{14}\,\left(1^7\right)$$

<div align="center">Type: SymmetricPolynomial Fraction Integer</div>

The operation `graphs` for argument `n` returns the cycle index of the group of permutations on the edges of the complete graph with `n` nodes induced by applying the symmetric group to the nodes.

`graphs 5`

$$\tfrac{1}{6}\,\left(6\ \ 3\ \ 1\right) + \tfrac{1}{5}\,\left(5^2\right) + \tfrac{1}{4}\,\left(4^2\ 2\right) + \tfrac{1}{6}\,\left(3^3\ 1\right) + \tfrac{1}{8}\,\left(2^4\ 1^2\right)+$$

$$\frac{1}{12}\,\left(2^3\ 1^4\right) + \frac{1}{120}\,\left(1^{10}\right)$$

<div align="center">Type: SymmetricPolynomial Fraction Integer</div>

The cycle index of a direct product of two groups is the product of the cycle indices of the groups. Redfield provided two operations on two cycle indices which will be called "cup" and "cap" here. The `cup` of two cycle indices is a kind of scalar product that combines monomials for permutations with the same cycles. The `cap` operation provides the sum of the coefficients of the result of the `cup` operation which will be an integer that enumerates what Redfield called group-reduced distributions.

We can, for example, represent `complete 2 * complete 2` as the set of objects `a a b b` and `complete 2 * complete 1 * complete 1` as `c c d e`.

This integer is the number of different sets of four pairs.

```
cap(complete 2**2, complete 2*complete 1**2)
```

$$4$$

                                                Type: Fraction Integer

For example,

```
a a b b    a a b b    a a b b    a a b b
c c d e    c d c e    c e c d    d e c c
```

This integer is the number of different sets of four pairs no two pairs being equal.

```
cap(elementary 2**2, complete 2*complete 1**2)
```

$$2$$

                                                Type: Fraction Integer

For example,

```
a a b b    a a b b
c d c e    c e c d
```

In this case the configurations enumerated are easily constructed, however the theory merely enumerates them providing little help in actually constructing them.

Here are the number of 6-pairs, first from `a a a b b c,` second from `d d e e f g.`

```
cap(complete 3*complete 2*complete 1,complete 2**2*complete 1**2)
```

$$24$$

                                                Type: Fraction Integer

Here it is again, but with no equal pairs.

```
cap(elementary 3*elementary 2*elementary 1,complete 2**2*complete
1**2)
```

$$8$$

<div align="right">Type: Fraction Integer</div>

```
cap(complete 3*complete 2*complete 1,elementary 2**2*elementary
1**2)
```

<div align="center">8</div>

<div align="right">Type: Fraction Integer</div>

The number of 6-triples, first from `a a a b b c`, second from `d d e e f g`, third from `h h i i j j`.

```
eval(cup(complete 3*complete 2*complete 1, cup(complete
2**2*complete 1**2,complete 2**3)))
```

<div align="center">1500</div>

<div align="right">Type: Fraction Integer</div>

The cycle index of vertices of a square is dihedral 4.

```
square:=dihedral 4
```

$$\frac{1}{4}\left(4\right)+\frac{3}{8}\left(2^2\right)+\frac{1}{4}\left(2\ 1^2\right)+\frac{1}{8}\left(1^4\right)$$

<div align="center">Type: SymmetricPolynomial Fraction Integer</div>

The number of different squares with 2 red vertices and 2 blue vertices.

```
cap(complete 2**2,square)
```

<div align="center">2</div>

<div align="right">Type: Fraction Integer</div>

The number of necklaces with 3 red beads, 2 blue beads and 2 green beads.

```
cap(complete 3*complete 2**2,dihedral 7)
```

<div align="center">18</div>

The number of graphs with 5 nodes and 7 edges.

```
cap(graphs 5,complete 7*complete 3)
```

$$4$$

Type: Fraction Integer

The cycle index of rotations of vertices of a cube.

```
s(x) == powerSum(x)
```

Type: Void

```
cube:=(1/24)*(s 1**8+9*s 2**4 + 8*s 3**2*s 1**2+6*s 4**2)
```

```
   Compiling function s with type PositiveInteger ->
      SymmetricPolynomial Fraction Integer
```

$$\frac{1}{4}\left(4^2\right) + \frac{1}{3}\left(3^2\ 1^2\right) + \frac{3}{8}\left(2^4\right) + \frac{1}{24}\left(1^8\right)$$

Type: SymmetricPolynomial Fraction Integer

The number of cubes with 4 red vertices and 4 blue vertices.

```
cap(complete 4**2,cube)
```

$$7$$

Type: Fraction Integer

The number of labeled graphs with degree sequence 2 2 2 1 1 with no loops
or multiple edges.

```
cap(complete 2**3*complete 1**2,wreath(elementary 4,elementary
2))
```

$$7$$

                                                    Type: Fraction Integer

Again, but with loops allowed but not multiple edges.

```
cap(complete 2**3*complete 1**2,wreath(elementary 4,complete 2))
```

                            17

                                                    Type: Fraction Integer

Again, but with multiple edges allowed, but not loops

```
cap(complete 2**3*complete 1**2,wreath(complete 4,elementary 2))
```

                            10

                                                    Type: Fraction Integer

Again, but with both multiple edges and loops allowed

```
cap(complete 2**3*complete 1**2,wreath(complete 4,complete 2))
```

                            23

                                                    Type: Fraction Integer

Having constructed a cycle index for a configuration we are at liberty to evaluate the s_i components any way we please. For example we can produce enumerating generating functions. This is done by providing a function `f` on an integer `i` to the value required of s_i, and then evaluating `eval(f, cycleindex)`.

```
x:  ULS(FRAC INT,'x,0) := 'x
```

                            x

                Type: UnivariateLaurentSeries(Fraction Integer,x,0)

```
ZeroOrOne:  INT -> ULS(FRAC INT, 'x, 0)
```

                                                    Type: Void

```
Integers:  INT -> ULS(FRAC INT, 'x, 0)
```

                                                    Type: Void

For the integers 0 and 1, or two colors.

```
ZeroOrOne n == 1+x**n
```

                                                    Type: Void

```
ZeroOrOne 5
```

```
   Compiling function ZeroOrOne with type Integer ->
      UnivariateLaurentSeries(Fraction Integer,x,0)
```

$$1 + x^5$$

          Type: UnivariateLaurentSeries(Fraction Integer,x,0)

For the integers `0, 1, 2, ...` we have this.

```
Integers n == 1/(1-x**n)
```

                                                    Type: Void

```
Integers 5
```

```
   Compiling function Integers with type Integer ->
      UnivariateLaurentSeries(Fraction Integer,x,0)
```

$$1 + x^5 + O\left(x^8\right)$$

          Type: UnivariateLaurentSeries(Fraction Integer,x,0)

The coefficient of x^n is the number of graphs with 5 nodes and **n** edges.

Note that there is an eval function that takes two arguments. It has the signature:

```
((Integer -> D1),SymmetricPolynomial Fraction Integer) -> D1
  from EvaluateCycleIndicators D1 if D1 has ALGEBRA FRAC INT
```

This function is not normally exposed (it will not normally be considered in the list of eval functions) as it is only useful for this particular domain. To use it we ask that it be considered thus:

```
)expose EVALCYC
```

and now we can use it:

```
eval(ZeroOrOne, graphs 5)
```

$$1 + x + 2\ x^2 + 4\ x^3 + 6\ x^4 + 6\ x^5 + 6\ x^6 + 4\ x^7 + O\left(x^8\right)$$

Type: UnivariateLaurentSeries(Fraction Integer,x,0)

The coefficient of x^n is the number of necklaces with **n** red beads and **n-8** green beads.

```
eval(ZeroOrOne,dihedral 8)
```

$$1 + x + 4\ x^2 + 5\ x^3 + 8\ x^4 + 5\ x^5 + 4\ x^6 + x^7 + O\left(x^8\right)$$

Type: UnivariateLaurentSeries(Fraction Integer,x,0)

The coefficient of x^n is the number of partitions of **n** into 4 or fewer parts.

```
eval(Integers,complete 4)
```

$$1 + x + 2\ x^2 + 3\ x^3 + 5\ x^4 + 6\ x^5 + 9\ x^6 + 11\ x^7 + O\left(x^8\right)$$

Type: UnivariateLaurentSeries(Fraction Integer,x,0)

The coefficient of x^n is the number of partitions of **n** into 4 boxes containing ordered distinct parts.

```
eval(Integers,elementary 4)
```

$$x^6 + x^7 + 2\ x^8 + 3\ x^9 + 5\ x^{10} + 6\ x^{11} + 9\ x^{12} + 11\ x^{13} + O\left(x^{14}\right)$$

Type: UnivariateLaurentSeries(Fraction Integer,x,0)

The coefficient of x^n is the number of different cubes with $\mathtt{n}$ red vertices and $\mathtt{8\text{-}n}$ green ones.

`eval(ZeroOrOne,cube)`

$$1 + x + 3\ x^2 + 3\ x^3 + 7\ x^4 + 3\ x^5 + 3\ x^6 + x^7 + O\left(x^8\right)$$

Type: UnivariateLaurentSeries(Fraction Integer,x,0)

The coefficient of x^n is the number of different cubes with integers on the vertices whose sum is $\mathtt{n}$.

`eval(Integers,cube)`

$$1 + x + 4\ x^2 + 7\ x^3 + 21\ x^4 + 37\ x^5 + 85\ x^6 + 151\ x^7 + O\left(x^8\right)$$

Type: UnivariateLaurentSeries(Fraction Integer,x,0)

The coefficient of x^n is the number of graphs with 5 nodes and with integers on the edges whose sum is $\mathtt{n}$. In other words, the enumeration is of multigraphs with 5 nodes and $\mathtt{n}$ edges.

`eval(Integers,graphs 5)`

$$1 + x + 3\ x^2 + 7\ x^3 + 17\ x^4 + 35\ x^5 + 76\ x^6 + 149\ x^7 + O\left(x^8\right)$$

Type: UnivariateLaurentSeries(Fraction Integer,x,0)

Graphs with 15 nodes enumerated with respect to number of edges.

`eval(ZeroOrOne ,graphs 15)`

$$1 + x + 2\ x^2 + 5\ x^3 + 11\ x^4 + 26\ x^5 + 68\ x^6 + 177\ x^7 + O\left(x^8\right)$$

Type: UnivariateLaurentSeries(Fraction Integer,x,0)

Necklaces with 7 green beads, 8 white beads, 5 yellow beads and 10 red beads.

`cap(dihedral 30,complete 7*complete 8*complete 5*complete 10)`

$$49958972383320$$

The operation `SFunction` is the S-function or Schur function of a partition written as a descending list of integers expressed in terms of power sum symmetric functions.

In this case the argument partition represents a tableau shape. For example `3,2,2,1` represents a tableau with three boxes in the first row, two boxes in the second and third rows, and one box in the fourth row. `SFunction [3,2,2,1]` counts the number of different tableaux of shape `3, 2, 2, 1` filled with objects with an ascending order in the columns and a non-descending order in the rows.

```
sf3221:= SFunction [3,2,2,1]
```

$$\frac{1}{12}\left(6\ 2\right) - \frac{1}{12}\left(6\ 1^2\right) - \frac{1}{16}\left(4^2\right) + \frac{1}{12}\left(4\ 3\ 1\right) + \frac{1}{24}\left(4\ 1^4\right) - \frac{1}{36}\left(3^2\ 2\right) +$$

$$\frac{1}{36}\left(3^2\ 1^2\right) - \frac{1}{24}\left(3\ 2^2\ 1\right) - \frac{1}{36}\left(3\ 2\ 1^3\right) - \frac{1}{72}\left(3\ 1^5\right) - \frac{1}{192}\left(2^4\right) +$$

$$\frac{1}{48}\left(2^3\ 1^2\right) + \frac{1}{96}\left(2^2\ 1^4\right) - \frac{1}{144}\left(2\ 1^6\right) + \frac{1}{576}\left(1^8\right)$$

Type: SymmetricPolynomial Fraction Integer

This is the number filled with `a a b b c c d d`.

```
cap(sf3221,complete 2**4)
```

$$3$$

Type: Fraction Integer

The configurations enumerated above are:

```
a a b     a a c     a a d
b c       b b       b b
c d       c d       c c
d         d         d
```

This is the number of tableaux filled with `1..8`.

```
cap(sf3221, powerSum 1**8)
```

$$70$$

Type: Fraction Integer

The coefficient of x^n is the number of column strict reverse plane partitions of
`n` of shape `3 2 2 1`.

`eval(Integers, sf3221)`

$$x^9 + 3\ x^{10} + 7\ x^{11} + 14\ x^{12} + 27\ x^{13} + 47\ x^{14} + O\left(x^{15}\right)$$

> Type: UnivariateLaurentSeries(Fraction Integer,x,0)

The smallest is

```
0 0 0
1 1
2 2
3
```

9.14   DeRhamComplex

The domain constructor `DeRhamComplex` creates the class of differential forms
of arbitrary degree over a coefficient ring. The De Rham complex constructor
takes two arguments: a ring, `coefRing,` and a list of coordinate variables.

This is the ring of coefficients.

`coefRing := Integer`

$$Integer$$

> Type: Domain

These are the coordinate variables.

`lv :   List Symbol := [x,y,z]`

$$[x, y, z]$$

> Type: List Symbol

This is the De Rham complex of Euclidean three-space using coordinates `x`, `y`
and `z`.

`der := DERHAM(coefRing,lv)`

$$DeRhamComplex(Integer, [x, y, z])$$

<div align="right">Type: Domain</div>

This complex allows us to describe differential forms having expressions of integers as coefficients. These coefficients can involve any number of variables, for example, `f(x,t,r,y,u,z)`. As we've chosen to work with ordinary Euclidean three-space, expressions involving these forms are treated as functions of `x`, `y` and `z` with the additional arguments `t`, `r` and `u` regarded as symbolic constants.

Here are some examples of coefficients.

```
R := Expression coefRing
```

<div align="center">Expression Integer</div>

<div align="right">Type: Domain</div>

```
f :  R := x**2*y*z-5*x**3*y**2*z**5
```

$$-5\ x^3\ y^2\ z^5 + x^2\ y\ z$$

<div align="right">Type: Expression Integer</div>

```
g :  R := z**2*y*cos(z)-7*sin(x**3*y**2)*z**2
```

$$-7\ z^2\ \sin\left(x^3\ y^2\right) + y\ z^2\ \cos\left(z\right)$$

<div align="right">Type: Expression Integer</div>

```
h :  R :=x*y*z-2*x**3*y*z**2
```

$$-2\ x^3\ y\ z^2 + x\ y\ z$$

<div align="right">Type: Expression Integer</div>

We now define the multiplicative basis elements for the exterior algebra over `R`.

```
dx :  der := generator(1)
```

$$dx$$

$$\text{Type:}\quad \texttt{DeRhamComplex(Integer,[x,y,z])}$$

```
dy :  der := generator(2)
```

$$dy$$

$$\text{Type:}\quad \texttt{DeRhamComplex(Integer,[x,y,z])}$$

```
dz :  der := generator(3)
```

$$dz$$

$$\text{Type:}\quad \texttt{DeRhamComplex(Integer,[x,y,z])}$$

This is an alternative way to give the above assignments.

```
[dx,dy,dz] := [generator(i)$der for i in 1..3]
```

$$[dx, dy, dz]$$

$$\text{Type:}\quad \texttt{List DeRhamComplex(Integer,[x,y,z])}$$

Now we define some one-forms.

```
alpha :  der := f*dx + g*dy + h*dz
```

$$\left(-2\ x^3\ y\ z^2 + x\ y\ z\right)\ dz +$$

$$\left(-7\ z^2\ \sin\left(x^3\ y^2\right) + y\ z^2\ \cos\left(z\right)\right)\ dy +$$

$$\left(-5\ x^3\ y^2\ z^5 + x^2\ y\ z\right)\ dx$$

$$\text{Type:}\quad \texttt{DeRhamComplex(Integer,[x,y,z])}$$

```
beta :  der := cos(tan(x*y*z)+x*y*z)*dx + x*dy
```

$$x\ dy + \cos\left(\tan\left(x\ y\ z\right) + x\ y\ z\right)\ dx$$

$$\text{Type:}\quad \texttt{DeRhamComplex(Integer,[x,y,z])}$$

A well-known theorem states that the composition of **exteriorDifferential** with itself is the zero map for continuous forms. Let's verify this theorem for `alpha`.

`exteriorDifferential alpha`

$$\left(y \ z^2 \ \sin\left(z\right) + 14 \ z \ \sin\left(x^3 \ y^2\right) - 2 \ y \ z \ \cos\left(z\right) - 2 \ x^3 \ z^2 + x \ z\right) \ dy \ dz+$$

$$\left(25 \ x^3 \ y^2 \ z^4 - 6 \ x^2 \ y \ z^2 + y \ z - x^2 \ y\right) \ dx \ dz+$$

$$\left(-21 \ x^2 \ y^2 \ z^2 \ \cos\left(x^3 \ y^2\right) + 10 \ x^3 \ y \ z^5 - x^2 \ z\right) \ dx \ dy$$

$$\text{Type:} \quad \text{DeRhamComplex(Integer,[x,y,z])}$$

We see a lengthy output of the last expression, but nevertheless, the composition is zero.

`exteriorDifferential %`

$$0$$

$$\text{Type:} \quad \text{DeRhamComplex(Integer,[x,y,z])}$$

Now we check that **exteriorDifferential** is a "graded derivation" `D`, that is, `D` satisfies:

`D(a*b) = D(a)*b + (-1)**degree(a)*a*D(b)`

`gamma := alpha * beta`

$$\left(2 \ x^4 \ y \ z^2 - x^2 \ y \ z\right) \ dy \ dz+$$

$$\left(2 \ x^3 \ y \ z^2 - x \ y \ z\right) \ \cos\left(\tan\left(x \ y \ z\right) + x \ y \ z\right) \ dx \ dz+$$

$$\left(\left(7 \ z^2 \ \sin\left(x^3 \ y^2\right) - y \ z^2 \ \cos\left(z\right)\right) \ \cos\left(\tan\left(x \ y \ z\right) + x \ y \ z\right)-\right.$$

$$\left. 5 \ x^4 \ y^2 \ z^5 + x^3 \ y \ z\right) \ dx \ dy$$

$$\text{Type:} \quad \text{DeRhamComplex(Integer,[x,y,z])}$$

We try this for the one-forms `alpha` and `beta`.

```
exteriorDifferential(gamma) - (exteriorDifferential(alpha)*beta -
alpha * exteriorDifferential(beta))
```

$$0$$

                              Type: DeRhamComplex(Integer,[x,y,z])

Now we define some "basic operators" (see 9.58 on page 717).

```
a :  BOP := operator('a)
```

$$a$$

                                          Type: BasicOperator

```
b :  BOP := operator('b)
```

$$b$$

                                          Type: BasicOperator

```
c :  BOP := operator('c)
```

$$c$$

                                          Type: BasicOperator

We also define some indeterminate one- and two-forms using these operators.

```
sigma := a(x,y,z) * dx + b(x,y,z) * dy + c(x,y,z) * dz
```

$$c\,(x,y,z)\ dz + b\,(x,y,z)\ dy + a\,(x,y,z)\ dx$$

                          Type: DeRhamComplex(Integer,[x,y,z])

```
theta := a(x,y,z) * dx * dy + b(x,y,z) * dx * dz + c(x,y,z) * dy
* dz
```

$$c\,(x,y,z)\ dy\ dz + b\,(x,y,z)\ dx\ dz + a\,(x,y,z)\ dx\ dy$$

                          Type: DeRhamComplex(Integer,[x,y,z])

This allows us to get formal definitions for the "gradient" ...

```
totalDifferential(a(x,y,z))$der
```

$$a_{,3}(x,y,z) \ dz + a_{,2}(x,y,z) \ dy + a_{,1}(x,y,z) \ dx$$

<div align="center">Type: DeRhamComplex(Integer,[x,y,z])</div>

the "curl" ...

```
exteriorDifferential sigma
```

$$(c_{,2}(x,y,z) - b_{,3}(x,y,z)) \ dy \ dz+$$

$$(c_{,1}(x,y,z) - a_{,3}(x,y,z)) \ dx \ dz+$$

$$(b_{,1}(x,y,z) - a_{,2}(x,y,z)) \ dx \ dy$$

<div align="center">Type: DeRhamComplex(Integer,[x,y,z])</div>

and the "divergence."

```
exteriorDifferential theta
```

$$(c_{,1}(x,y,z) - b_{,2}(x,y,z) + a_{,3}(x,y,z)) \ dx \ dy \ dz$$

<div align="center">Type: DeRhamComplex(Integer,[x,y,z])</div>

Note that the De Rham complex is an algebra with unity. This element 1 is the basis for elements for zero-forms, that is, functions in our space.

```
one :  der := 1
```

<div align="center">1</div>

<div align="center">Type: DeRhamComplex(Integer,[x,y,z])</div>

To convert a function to a function lying in the De Rham complex, multiply the function by "one."

```
g1 :  der := a([x,t,y,u,v,z,e]) * one
```

$$a(x,t,y,u,v,z,e)$$

<div align="center">Type: DeRhamComplex(Integer,[x,y,z])</div>

A current limitation of Axiom forces you to write functions with more than four arguments using square brackets in this way.

```
h1 :  der := a([x,y,x,t,x,z,y,r,u,x]) * one
```

$$a\,(x, y, x, t, x, z, y, r, u, x)$$

<div align="right">Type: DeRhamComplex(Integer,[x,y,z])</div>

Now note how the system keeps track of where your coordinate functions are located in expressions.

```
exteriorDifferential g1
```

$$a_{,6}\,(x, t, y, u, v, z, e)\ dz+$$

$$a_{,3}\,(x, t, y, u, v, z, e)\ dy+$$

$$a_{,1}\,(x, t, y, u, v, z, e)\ dx$$

<div align="right">Type: DeRhamComplex(Integer,[x,y,z])</div>

```
exteriorDifferential h1
```

$$a_{,6}\,(x, y, x, t, x, z, y, r, u, x)\ dz+$$

$$(a_{,7}\,(x, y, x, t, x, z, y, r, u, x)+$$

$$a_{,2}\,(x, y, x, t, x, z, y, r, u, x))\ dy+$$

$$(a_{,10}\,(x, y, x, t, x, z, y, r, u, x)+$$

$$a_{,5}\,(x, y, x, t, x, z, y, r, u, x)+$$

$$a_{,3}\,(x, y, x, t, x, z, y, r, u, x)+$$

$$a_{,1}\,(x, y, x, t, x, z, y, r, u, x))\ dx$$

<div align="right">Type: DeRhamComplex(Integer,[x,y,z])</div>

In this example of Euclidean three-space, the basis for the De Rham complex consists of the eight forms: `1`, `dx`, `dy`, `dz`, `dx*dy`, `dx*dz`, `dy*dz`, and `dx*dy*dz`.

```
coefficient(gamma, dx*dy)
```

$$\left(7\ z^2\ \sin\left(x^3\ y^2\right) - y\ z^2\ \cos\left(z\right)\right)\ \cos\left(\tan\left(x\ y\ z\right) + x\ y\ z\right)$$

$$-5\ x^4\ y^2\ z^5 + x^3\ y\ z$$

<div align="right">Type: Expression Integer</div>

coefficient(gamma, one)

$$0$$

<div align="right">Type: Expression Integer</div>

coefficient(g1,one)

$$a\left(x, t, y, u, v, z, e\right)$$

<div align="right">Type: Expression Integer</div>

9.15   DecimalExpansion

All rationals have repeating decimal expansions. Operations to access the individual digits of a decimal expansion can be obtained by converting the value to `RadixExpansion(10)`. More examples of expansions are available in 9.4 on page 468, 9.33 on page 586, and 9.65 on page 747.

The operation **decimal** is used to create this expansion of type `DecimalExpansion`.

r := decimal(22/7)

$$3.\overline{142857}$$

<div align="right">Type: DecimalExpansion</div>

Arithmetic is exact.

r + decimal(6/7)

$$4$$

<div align="right">Type: DecimalExpansion</div>

The period of the expansion can be short or long . . .

```
[decimal(1/i) for i in 350..354]
```

$$\left[0.00\overline{285714}, 0.00\overline{2849}, 0.002840\overline{9}, 0.00\overline{28328611898016997167138810198\overline{3}},\right.$$

$$\left.0.00\overline{2824858757062146892655367231638418079096045197740112994\overline{35}}\right]$$

<div align="right">Type: List DecimalExpansion</div>

or very long.

```
decimal(1/2049)
```

$$0.000488042947779404587603709126403123474865788189360663738408979990239$$
$$\overline{14104441190824792581747193753050268423621278672523182040019521717911\overline{1}}$$
$$\overline{76183504148365056124938994631527574426549536359199609565641776476329\overline{9}}$$
$$\overline{17032698877501220107369448511469009272816007808687164470473401659346\overline{0}}$$
$$\overline{2244997559785261102977061981454367984382625671059053196681307955\overline{1}}$$

<div align="right">Type: DecimalExpansion</div>

These numbers are bona fide algebraic objects.

```
p := decimal(1/4)*x**2 + decimal(2/3)*x + decimal(4/9)
```

$$0.25\ x^2 + 0.\overline{6}\ x + 0.\overline{4}$$

<div align="right">Type: Polynomial DecimalExpansion</div>

```
q := differentiate(p, x)
```

$$0.5\ x + 0.\overline{6}$$

<div align="right">Type: Polynomial DecimalExpansion</div>

```
g := gcd(p, q)
```

$$x + 1.\overline{3}$$

<div align="right">Type: Polynomial DecimalExpansion</div>

9.16 DistributedMultivariatePolynomial

`DistributedMultivariatePolynomial` which is abbreviated as `DMP` and
`HomogeneousDistributedMultivariatePolynomial`, which is abbreviated as
`HDMP`, are very similar to `MultivariatePolynomial` except that they are repre-
sented and displayed in a non-recursive manner.

```
(d1,d2,d3) :  DMP([z,y,x],FRAC INT)
```

                                                            Type: Void

The constructor `DMP` orders its monomials lexicographically while `HDMP` orders
them by total order refined by reverse lexicographic order.

```
d1 := -4*z + 4*y**2*x + 16*x**2 + 1
```

$$-4\ z + 4\ y^2\ x + 16\ x^2 + 1$$

        Type: DistributedMultivariatePolynomial([z,y,x],Fraction
                                                            Integer)

```
d2 := 2*z*y**2 + 4*x + 1
```

$$2\ z\ y^2 + 4\ x + 1$$

        Type: DistributedMultivariatePolynomial([z,y,x],Fraction
                                                            Integer)

```
d3 := 2*z*x**2 - 2*y**2 - x
```

$$2\ z\ x^2 - 2\ y^2 - x$$

        Type: DistributedMultivariatePolynomial([z,y,x],Fraction
                                                            Integer)

These constructors are mostly used in Gröbner basis calculations.

```
groebner [d1,d2,d3]
```

$$\Big[z - \tfrac{1568}{2745}\, x^6 - \tfrac{1264}{305}\, x^5 + \tfrac{6}{305}\, x^4 + \tfrac{182}{549}\, x^3 - \tfrac{2047}{610}\, x^2 - \tfrac{103}{2745}\, x - \tfrac{2857}{10980},$$

$$y^2 + \frac{112}{2745}\, x^6 - \frac{84}{305}\, x^5 - \frac{1264}{305}\, x^4 - \frac{13}{549}\, x^3 + \frac{84}{305}\, x^2 + \frac{1772}{2745}\, x + \frac{2}{2745},$$

$$x^7 + \frac{29}{4}\, x^6 - \frac{17}{16}\, x^4 - \frac{11}{8}\, x^3 + \frac{1}{32}\, x^2 + \frac{15}{16}\, x + \frac{1}{4}\Big]$$

```
    Type:  List DistributedMultivariatePolynomial([z,y,x],Fraction
                                                              Integer)
```

```
(n1,n2,n3) :  HDMP([z,y,x],FRAC INT)
```

```
                                                          Type:  Void
```

```
n1 := d1
```

$$4\, y^2\, x + 16\, x^2 - 4\, z + 1$$

```
                                                        Type:
    HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction
                                                              Integer)
```

```
n2 := d2
```

$$2\, z\, y^2 + 4\, x + 1$$

```
                                                        Type:
    HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction
                                                              Integer)
```

```
n3 := d3
```

$$2\, z\, x^2 - 2\, y^2 - x$$

```
                                                        Type:
    HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction
                                                              Integer)
```

Note that we get a different Gröbner basis when we use the `HDMP` polynomials, as expected.

```
groebner [n1,n2,n3]
```

$$\left[y^4 + 2\ x^3 - \tfrac{3}{2}\ x^2 + \tfrac{1}{2}\ z - \tfrac{1}{8},\right.$$

$$x^4 + \frac{29}{4}\ x^3 - \frac{1}{8}\ y^2 - \frac{7}{4}\ z\ x - \frac{9}{16}\ x - \frac{1}{4},$$

$$z\ y^2 + 2\ x + \frac{1}{2},$$

$$y^2\ x + 4\ x^2 - z + \frac{1}{4},$$

$$z\ x^2 - y^2 - \frac{1}{2}\ x,$$

$$\left. z^2 - 4\ y^2 + 2\ x^2 - \frac{1}{4}\ z - \frac{3}{2}\ x\right]$$

<div align="right">

Type: List
HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction
Integer)

</div>

`GeneralDistributedMultivariatePolynomial` is somewhat more flexible in the sense that as well as accepting a list of variables to specify the variable ordering, it also takes a predicate on exponent vectors to specify the term ordering. With this polynomial type the user can experiment with the effect of using completely arbitrary term orderings. This flexibility is mostly important for algorithms such as Gröbner basis calculations which can be very sensitive to term ordering.

9.17   DoubleFloat

Axiom provides two kinds of floating point numbers. The domain `Float` (abbreviation `FLOAT`) implements a model of arbitrary precision floating point numbers. The domain `DoubleFloat` (abbreviation `DFLOAT`) is intended to make available hardware floating point arithmetic in Axiom. The actual model of floating point `DoubleFloat` that provides is system-dependent. For example, on the IBM system 370 Axiom uses IBM double precision which has fourteen hexadecimal digits of precision or roughly sixteen decimal digits. Arbitrary precision floats allow the user to specify the precision at which arithmetic operations are computed. Although this is an attractive facility, it comes at a cost. Arbitrary-precision

floating-point arithmetic typically takes twenty to two hundred times more time than hardware floating point.

The usual arithmetic and elementary functions are available for `DoubleFloat`. Use `)show DoubleFloat` to get a list of operations or the HyperDoc browse facility to get more extensive documentation about `DoubleFloat`.

By default, floating point numbers that you enter into Axiom are of type `Float`.

```
2.71828
```

$$2.71828$$

<div align="right">

Type: Float

</div>

You must therefore tell Axiom that you want to use `DoubleFloat` values and operations. The following are some conservative guidelines for getting Axiom to use `DoubleFloat`.

To get a value of type `DoubleFloat`, use a target with `@`, ...

```
2.71828@DoubleFloat
```

$$2.71828$$

<div align="right">

Type: DoubleFloat

</div>

a conversion, ...

```
2.71828 ::  DoubleFloat
```

$$2.71828$$

<div align="right">

Type: DoubleFloat

</div>

or an assignment to a declared variable. It is more efficient if you use a target rather than an explicit or implicit conversion.

```
eApprox :  DoubleFloat := 2.71828
```

$$2.71828$$

<div align="right">

Type: DoubleFloat

</div>

You also need to declare functions that work with `DoubleFloat`.

```
avg :  List DoubleFloat -> DoubleFloat
```

<div align="right">Type: Void</div>

```
avg l ==
  empty? l => 0 :: DoubleFloat
  reduce(_+,l) / #l
```

<div align="right">Type: Void</div>

```
avg [3.4,9.7,-6.8]
```

```
   Compiling function avg with type List Float -> DoubleFloat
```

<div align="center">2.1</div>

<div align="right">Type: DoubleFloat</div>

Use package-calling for operations from `DoubleFloat` unless the arguments themselves are already of type `DoubleFloat`.

```
cos(3.1415926)$DoubleFloat
```

<div align="center">-0.999999999999999</div>

<div align="right">Type: DoubleFloat</div>

```
cos(3.1415926 ::  DoubleFloat)
```

<div align="center">-0.999999999999999</div>

<div align="right">Type: DoubleFloat</div>

By far, the most common usage of `DoubleFloat` is for functions to be graphed. For more information about Axiom's numerical and graphical facilities, see Section 7 on page 297, 8.1 on page 345, and 9.27 on page 565.

9.18   EqTable

The `EqTable` domain provides tables where the keys are compared using **eq?**. Keys are considered equal only if they are the same instance of a structure. This is useful if the keys are themselves updatable structures. Otherwise, all operations are the same as for type `Table`. See 9.80 on page 816 for general information about tables.

The operation **table** is here used to create a table where the keys are lists of integers.

```
e:  EqTable(List Integer, Integer) := table()
```

$$table()$$

```
                                    Type:  EqTable(List Integer,Integer)
```

These two lists are equal according to "=", but not according to **eq?**.

```
l1 := [1,2,3]
```

$$[1, 2, 3]$$

```
                                          Type:  List PositiveInteger
```

```
l2 := [1,2,3]
```

$$[1, 2, 3]$$

```
                                          Type:  List PositiveInteger
```

Because the two lists are not **eq?**, separate values can be stored under each.

```
e.l1 := 111
```

$$111$$

```
                                          Type:  PositiveInteger
```

```
e.l2 := 222
```

$$222$$

e.l1

$$111$$

Type: PositiveInteger

9.19   Equation

The `Equation` domain provides equations as mathematical objects. These are used, for example, as the input to various **solve** operations.

Equations are created using the equals symbol, "=".

eq1 := 3*x + 4*y = 5

$$4\ y + 3\ x = 5$$

Type: Equation Polynomial Integer

eq2 := 2*x + 2*y = 3

$$2\ y + 2\ x = 3$$

Type: Equation Polynomial Integer

The left- and right-hand sides of an equation are accessible using the operations **lhs** and **rhs**.

lhs eq1

$$4\ y + 3\ x$$

Type: Polynomial Integer

rhs eq1

<div align="right">Type: Polynomial Integer</div>

Arithmetic operations are supported and operate on both sides of the equation.

`eq1 + eq2`

$$6\ y + 5\ x = 8$$

<div align="right">Type: Equation Polynomial Integer</div>

`eq1 * eq2`

$$8\ y^2 + 14\ x\ y + 6\ x^2 = 15$$

<div align="right">Type: Equation Polynomial Integer</div>

`2*eq2 - eq1`

$$x = 1$$

<div align="right">Type: Equation Polynomial Integer</div>

Equations may be created for any type so the arithmetic operations will be defined only when they make sense. For example, exponentiation is not defined for equations involving non-square matrices.

`eq1**2`

$$16\ y^2 + 24\ x\ y + 9\ x^2 = 25$$

<div align="right">Type: Equation Polynomial Integer</div>

Note that an equals symbol is also used to *test* for equality of values in certain contexts. For example, `x+1` and `y` are unequal as polynomials.

`if x+1 = y then "equal" else "unequal"`

<div align="center">"unequal"</div>

<div align="right">Type: String</div>

```
eqpol := x+1 = y
```

$$x + 1 = y$$

                    Type: Equation Polynomial Integer

If an equation is used where a `Boolean` value is required, then it is evaluated using the equality test from the operand type.

```
if eqpol then "equal" else "unequal"
```

                    "unequal"

                              Type: String

If one wants a `Boolean` value rather than an equation, all one has to do is ask!

```
eqpol::Boolean
```

                    false

                              Type: Boolean

9.20 Exit

A function that does not return directly to its caller has `Exit` as its return type. The operation `error` is an example of one which does not return to its caller. Instead, it causes a return to top-level.

```
n := 0
```

                    0

                              Type: NonNegativeInteger

The function `gasp` is given return type `Exit` since it is guaranteed never to return a value to its caller.

```
gasp(): Exit ==
    free n
    n := n + 1
    error "Oh no!"
```

```
Function declaration gasp : () -> Exit has been added to workspace.
```

                                                           Type: Void

The return type of `half` is determined by resolving the types of the two branches
of the `if`.

```
half(k) ==
  if odd? k then gasp()
  else k quo 2
```

Because `gasp` has the return type `Exit`, the type of `if` in `half` is resolved to be
`Integer`.

`half 4`

```
   Compiling function gasp with type () -> Exit
   Compiling function half with type PositiveInteger -> Integer
```

                                2

                                                 Type: PositiveInteger

`half 3`

```
   Error signalled from user code in function gasp:
      Oh no!
```

n

                                1

                                            Type: NonNegativeInteger

For functions which return no value at all, use `Void`. See **??** on page **??** in
Section **??** on page **??** and 9.86 on page 839 for more information.

9.21   Expression

`Expression` is a constructor that creates domains whose objects can have very
general symbolic forms. Here are some examples:

This is an object of type `Expression Integer`.

```
sin(x) + 3*cos(x)**2
```

$$\sin{(x)} + 3 \cos{(x)}^2$$

<div align="right">Type: Expression Integer</div>

This is an object of type `Expression Float`.

```
tan(x) - 3.45*x
```

$$\tan{(x)} - 3.45 \ x$$

<div align="right">Type: Expression Float</div>

This object contains symbolic function applications, sums, products, square roots, and a quotient.

```
(tan sqrt 7 - sin sqrt 11)**2 / (4 - cos(x - y))
```

$$\frac{-\tan{\left(\sqrt{7}\right)}^2 + 2 \ \sin{\left(\sqrt{11}\right)} \ \tan{\left(\sqrt{7}\right)} - \sin{\left(\sqrt{11}\right)}^2}{\cos{(y - x)} - 4}$$

<div align="right">Type: Expression Integer</div>

As you can see, `Expression` actually takes an argument domain. The *coefficients* of the terms within the expression belong to the argument domain. `Integer` and `Float`, along with `Complex Integer` and `Complex Float` are the most common coefficient domains.

The choice of whether to use a `Complex` coefficient domain or not is important since Axiom can perform some simplifications on real-valued objects

```
log(exp x)@Expression(Integer)
```

$$x$$

<div align="right">Type: Expression Integer</div>

... which are not valid on complex ones.

```
log(exp x)@Expression(Complex Integer)
```

$$\log{(e^x)}$$

                                         Type: Expression Complex Integer

Many potential coefficient domains, such as `AlgebraicNumber`, are not usually
used because `Expression` can subsume them.

```
sqrt 3 + sqrt(2 + sqrt(-5))
```

$$\sqrt{\sqrt{-5}+2} + \sqrt{3}$$

                                         Type: AlgebraicNumber

```
% :: Expression Integer
```

$$\sqrt{\sqrt{-5}+2} + \sqrt{3}$$

                                         Type: Expression Integer

Note that we sometimes talk about "an object of type `Expression`." This
is not really correct because we should say, for example, "an object of type
`Expression Integer`" or "an object of type `Expression Float`." By a similar
abuse of language, when we refer to an "expression" in this section we will mean
an object of type `Expression R` for some domain `R`.

The Axiom documentation contains many examples of the use of `Expression`.
For the rest of this section, we'll give you some pointers to those examples plus
give you some idea of how to manipulate expressions.

It is important for you to know that `Expression` creates domains that have cat-
egory `Field`. Thus you can invert any non-zero expression and you shouldn't
expect an operation like `factor` to give you much information. You can imag-
ine expressions as being represented as quotients of "multivariate" polynomials
where the "variables" are kernels (see 9.37 on page 604). A kernel can either
be a symbol such as `x` or a symbolic function application like `sin(x + 4)`. The
second example is actually a nested kernel since the argument to `sin` contains
the kernel `x`.

```
height mainKernel sin(x + 4)
```

                                         2

                                         Type: PositiveInteger

Actually, the argument to `sin` is an expression, and so the structure of `Expression` is recursive. 9.37 on page 604 demonstrates how to extract the kernels in an expression.

Use the HyperDoc Browse facility to see what operations are applicable to expression. At the time of this writing, there were 262 operations with 147 distinct name in `Expression Integer`. For example, **numer** and **denom** extract the numerator and denominator of an expression.

```
e := (sin(x) - 4)**2 / ( 1 - 2*y*sqrt(- y) )
```

$$\frac{-\sin{(x)}^2 + 8\,\sin{(x)} - 16}{2\,y\,\sqrt{-y} - 1}$$

<div align="right">Type: Expression Integer</div>

```
numer e
```

$$-\sin{(x)}^2 + 8\,\sin{(x)} - 16$$

<div align="right">Type: SparseMultivariatePolynomial(Integer,Kernel Expression
Integer)</div>

```
denom e
```

$$2\,y\,\sqrt{-y} - 1$$

<div align="right">Type: SparseMultivariatePolynomial(Integer,Kernel Expression
Integer)</div>

Use **D** to compute partial derivatives.

```
D(e, x)
```

$$\frac{(4\,y\,\cos{(x)}\,\sin{(x)} - 16\,y\,\cos{(x)})\,\sqrt{-y} - 2\,\cos{(x)}\,\sin{(x)} + 8\,\cos{(x)}}{4\,y\,\sqrt{-y} + 4\,y^3 - 1}$$

<div align="right">Type: Expression Integer</div>

See **??** on page **??** in Section **??** on page **??** for more examples of expressions and derivatives.

```
D(e, [x, y], [1, 2])
```

$$\frac{\begin{pmatrix} \left(\left(-2304\ y^7 + 960\ y^4\right)\cos\left(x\right)\sin\left(x\right) + \left(9216\ y^7 - 3840\ y^4\right)\cos\left(x\right)\right)\sqrt{-y}+ \\[2mm] \left(-960\ y^9 + 2160\ y^6 - 180\ y^3 - 3\right)\cos\left(x\right)\sin\left(x\right)+ \\[2mm] \left(3840\ y^9 - 8640\ y^6 + 720\ y^3 + 12\right)\cos\left(x\right) \end{pmatrix}}{\begin{pmatrix} \left(256\ y^{12} - 1792\ y^9 + 1120\ y^6 - 112\ y^3 + 1\right)\sqrt{-y}- \\[2mm] 1024\ y^{11} + 1792\ y^8 - 448\ y^5 + 16\ y^2 \end{pmatrix}}$$

<div align="right">

Type: Expression Integer

</div>

See **??** on page **??** in Section **??** on page **??** and **??** on page **??** in Section **??** on page **??** for more examples of expressions and calculus. Differential equations involving expressions are discussed in **??** on page **??** in Section **??** on page **??**. Chapter 8 has many advanced examples: see **??** on page **??** in Section **??** on page **??** for a discussion of Axiom's integration facilities.

When an expression involves no "symbol kernels" (for example, `x`), it may be possible to numerically evaluate the expression.

If you suspect the evaluation will create a complex number, use `complexNumeric`.

`complexNumeric(cos(2 - 3*%i))`

$$-4.1896256909\ 688072301 + 9.1092278937\ 55336598\ i$$

<div align="right">

Type: Complex Float

</div>

If you know it will be real, use `numeric`.

`numeric(tan 3.8)`

$$0.7735560905\ 0312607286$$

<div align="right">

Type: Float

</div>

The `numeric` operation will display an error message if the evaluation yields a calue with an non-zero imaginary part. Both of these operations have an optional second argument `n` which specifies that the accuracy of the approximation be up to `n` decimal places.

When an expression involves no "symbolic application" kernels, it may be possible to convert it a polynomial or rational function in the variables that are present.

```
e2 := cos(x**2 - y + 3)
```

$$\cos\left(y - x^2 - 3\right)$$

<div align="right">Type: Expression Integer</div>

```
e3 := asin(e2) - %pi/2
```

$$-y + x^2 + 3$$

<div align="right">Type: Expression Integer</div>

```
e3 :: Polynomial Integer
```

$$-y + x^2 + 3$$

<div align="right">Type: Polynomial Integer</div>

This also works for the polynomial types where specific variables and their ordering are given.

```
e3 :: DMP([x, y], Integer)
```

$$x^2 - y + 3$$

<div align="center">Type: DistributedMultivariatePolynomial([x,y],Integer)</div>

Finally, a certain amount of simplication takes place as expressions are constructed.

```
sin %pi
```

$$0$$

<div align="right">Type: Expression Integer</div>

```
cos(%pi / 4)
```

$$\frac{\sqrt{2}}{2}$$

For simplications that involve multiple terms of the expression, use `simplify`.

`tan(x)**6 + 3*tan(x)**4 + 3*tan(x)**2 + 1`

$$\tan{(x)}^6 + 3\, \tan{(x)}^4 + 3\, \tan{(x)}^2 + 1$$

`simplify %`

$$\frac{1}{\cos{(x)}^6}$$

See **??** on page **??** in Section **??** on page **??** for examples of how to write your own rewrite rules for expressions.

9.22 Factored

`Factored` creates a domain whose objects are kept in factored form as long as possible. Thus certain operations like "`*`" (multiplication) and **gcd** are relatively easy to do. Others, such as addition, require somewhat more work, and the result may not be completely factored unless the argument domain `R` provides a **factor** operation. Each object consists of a unit and a list of factors, where each factor consists of a member of `R` (the *base*), an exponent, and a flag indicating what is known about the base. A flag may be one of "`nil`", "`sqfr`", "`irred`" or "`prime`", which mean that nothing is known about the base, it is square-free, it is irreducible, or it is prime, respectively. The current restriction to factored objects of integral domains allows simplification to be performed without worrying about multiplication order.

9.22.1 Decomposing Factored Objects

In this section we will work with a factored integer.

`g := factor(4312)`

$$2^3 \ 7^2 \ 11$$

<div align="right">Type: Factored Integer</div>

Let's begin by decomposing **g** into pieces. The only possible units for integers are `1` and `-1`.

`unit(g)`

<div align="center">1</div>

<div align="right">Type: PositiveInteger</div>

There are three factors.

`numberOfFactors(g)`

<div align="center">3</div>

<div align="right">Type: PositiveInteger</div>

We can make a list of the bases, . . .

`[nthFactor(g,i) for i in 1..numberOfFactors(g)]`

<div align="center">$[2, 7, 11]$</div>

<div align="right">Type: List Integer</div>

and the exponents, . . .

`[nthExponent(g,i) for i in 1..numberOfFactors(g)]`

<div align="center">$[3, 2, 1]$</div>

<div align="right">Type: List Integer</div>

and the flags. You can see that all the bases (factors) are prime.

`[nthFlag(g,i) for i in 1..numberOfFactors(g)]`

<div align="center">$\left[\texttt{"prime"}, \texttt{"prime"}, \texttt{"prime"}\right]$</div>

<div align="right">Type: List Union("nil","sqfr","irred","prime")</div>

A useful operation for pulling apart a factored object into a list of records of the components is **factorList**.

```
factorList(g)
```

$$[[flg = \texttt{"prime"}, fctr = 2, xpnt = 3],$$

$$[flg = \texttt{"prime"}, fctr = 7, xpnt = 2],$$

$$[flg = \texttt{"prime"}, fctr = 11, xpnt = 1]]$$

```
    Type:  List Record(flg:  Union("nil","sqfr","irred","prime"),
                                 fctr:  Integer,xpnt:  Integer)
```

If you don't care about the flags, use **factors**.

```
factors(g)
```

$$[[factor = 2, exponent = 3],$$

$$[factor = 7, exponent = 2],$$

$$[factor = 11, exponent = 1]]$$

```
        Type:  List Record(factor:  Integer,exponent:  Integer)
```

Neither of these operations returns the unit.

```
first(%).factor
```

$$2$$

```
                                        Type:  PositiveInteger
```

9.22.2   Expanding Factored Objects

Recall that we are working with this factored integer.

```
g := factor(4312)
```

$$2^3 \; 7^2 \; 11$$

```
                                        Type:  Factored Integer
```

To multiply out the factors with their multiplicities, use **expand**.

```
expand(g)
```

$$4312$$

<div align="right">Type: PositiveInteger</div>

If you would like, say, the distinct factors multiplied together but with multiplicity one, you could do it this way.

```
reduce(*,[t.factor for t in factors(g)])
```

$$154$$

<div align="right">Type: PositiveInteger</div>

9.22.3 Arithmetic with Factored Objects

We're still working with this factored integer.

```
g := factor(4312)
```

$$2^3 \; 7^2 \; 11$$

<div align="right">Type: Factored Integer</div>

We'll also define this factored integer.

```
f := factor(246960)
```

$$2^4 \; 3^2 \; 5 \; 7^3$$

<div align="right">Type: Factored Integer</div>

Operations involving multiplication and division are particularly easy with factored objects.

```
f * g
```

$$2^7 \; 3^2 \; 5 \; 7^5 \; 11$$

<div align="right">Type: Factored Integer</div>

`f**500`

$$2^{2000} \ 3^{1000} \ 5^{500} \ 7^{1500}$$

<div align="right">Type: Factored Integer</div>

`gcd(f,g)`

$$2^3 \ 7^2$$

<div align="right">Type: Factored Integer</div>

`lcm(f,g)`

$$2^4 \ 3^2 \ 5 \ 7^3 \ 11$$

<div align="right">Type: Factored Integer</div>

If we use addition and subtraction things can slow down because we may need to compute greatest common divisors.

`f + g`

$$2^3 \ 7^2 \ 641$$

<div align="right">Type: Factored Integer</div>

`f - g`

$$2^3 \ 7^2 \ 619$$

<div align="right">Type: Factored Integer</div>

Test for equality with 0 and 1 by using **zero?** and **one?**, respectively.

`zero?(factor(0))`

<div align="center">true</div>

                                                           Type: Boolean

zero?(g)

                                             false

                                                           Type: Boolean

one?(factor(1))

                                      true

                                                           Type: Boolean

one?(f)

                                             false

                                                           Type: Boolean

Another way to get the zero and one factored objects is to use package calling
(see **??** on page **??** in Section **??** on page **??**).

0$Factored(Integer)

                                        0

                                                   Type: Factored Integer

1$Factored(Integer)

                                        1

                                                   Type: Factored Integer

9.22.4   Creating New Factored Objects

The **map** operation is used to iterate across the unit and bases of a factored object. See **??** on page **??** for a discussion of **map**.

The following four operations take a base and an exponent and create a factored object. They differ in handling the flag component.

`nilFactor(24,2)`

$$24^2$$

<div align="right">Type: Factored Integer</div>

This factor has no associated information.

`nthFlag(%,1)`

<div align="center">"nil"</div>

<div align="right">Type: Union("nil",...)</div>

This factor is asserted to be square-free.

`sqfrFactor(30,2)`

$$30^2$$

<div align="right">Type: Factored Integer</div>

This factor is asserted to be irreducible.

`irreducibleFactor(13,10)`

$$13^{10}$$

<div align="right">Type: Factored Integer</div>

This factor is asserted to be prime.

`primeFactor(11,5)`

$$11^5$$

A partial inverse to **factorList** is **makeFR**.

```
h := factor(-720)
```

$$-2^4 \; 3^2 \; 5$$

<div align="right">Type: Factored Integer</div>

The first argument is the unit and the second is a list of records as returned by **factorList**.

```
h - makeFR(unit(h),factorList(h))
```

$$0$$

<div align="right">Type: Factored Integer</div>

9.22.5   Factored Objects with Variables

Some of the operations available for polynomials are also available for factored polynomials.

```
p := (4*x*x-12*x+9)*y*y + (4*x*x-12*x+9)*y + 28*x*x - 84*x + 63
```

$$\left(4 \; x^2 - 12 \; x + 9\right) \; y^2 + \left(4 \; x^2 - 12 \; x + 9\right) \; y + 28 \; x^2 - 84 \; x + 63$$

<div align="right">Type: Polynomial Integer</div>

```
fp := factor(p)
```

$$\left(2 \; x - 3\right)^2 \; \left(y^2 + y + 7\right)$$

<div align="right">Type: Factored Polynomial Integer</div>

You can differentiate with respect to a variable.

```
D(p,x)
```

$$\left(8 \; x - 12\right) \; y^2 + \left(8 \; x - 12\right) \; y + 56 \; x - 84$$

<div align="right">Type: Polynomial Integer</div>

```
D(fp,x)
```

$$4 \, (2 \, x - 3) \, \left(y^2 + y + 7\right)$$

<div align="right">Type: Factored Polynomial Integer</div>

```
numberOfFactors(%)
```

$$3$$

<div align="right">Type: PositiveInteger</div>

9.23   FactoredFunctions2

The `FactoredFunctions2` package implements one operation, **map**, for apply-
ing an operation to every base in a factored object and to the unit.

```
double(x) == x + x
```

<div align="right">Type: Void</div>

```
f := factor(720)
```

$$2^4 \, 3^2 \, 5$$

<div align="right">Type: Factored Integer</div>

Actually, the **map** operation used in this example comes from `Factored` itself,
since `double` takes an integer argument and returns an integer result.

```
map(double,f)
```

$$2 \, 4^4 \, 6^2 \, 10$$

<div align="right">Type: Factored Integer</div>

If we want to use an operation that returns an object that has a type different from the operation's argument, the **map** in `Factored` cannot be used and we use the one in `FactoredFunctions2`.

```
makePoly(b) == x + b
```

<div align="right">Type: Void</div>

In fact, the "2" in the name of the package means that we might be using factored objects of two different types.

```
g := map(makePoly,f)
```

$$(x + 1) \, (x + 2)^4 \, (x + 3)^2 \, (x + 5)$$

<div align="right">Type: Factored Polynomial Integer</div>

It is important to note that both versions of **map** destroy any information known about the bases (the fact that they are prime, for instance).

The flags for each base are set to "nil" in the object returned by **map**.

```
nthFlag(g,1)
```

<div align="center">"nil"</div>

<div align="right">Type: Union("nil",...)</div>

For more information about factored objects and their use, see 9.22 on page 546 and **??** on page **??** in Section **??** on page **??**.

9.24   File

The `File(S)` domain provides a basic interface to read and write values of type `S` in files.

Before working with a file, it must be made accessible to Axiom with the **open** operation.

```
ifile:File List Integer:=open("/tmp/jazz1","output")
```

<div align="center">"/tmp/jazz1"</div>

                                              Type: File List Integer

The **open** function arguments are a `FileName` and a `String` specifying the
mode. If a full pathname is not specified, the current default directory is as-
sumed. The mode must be one of "`input`" or "`output`". If it is not specified,
"`input`" is assumed. Once the file has been opened, you can read or write data.

The operations **read** and **write** are provided.

```
write!(ifile, [-1,2,3])
```

$$[-1, 2, 3]$$

                                              Type: List Integer

```
write!(ifile, [10,-10,0,111])
```

$$[10, -10, 0, 111]$$

                                              Type: List Integer

```
write!(ifile, [7])
```

$$[7]$$

                                              Type: List Integer

You can change from writing to reading (or vice versa) by reopening a file.

```
reopen!(ifile, "input")
```

                        "/tmp/jazz1"

                                              Type: File List Integer

```
read!  ifile
```

$$[-1, 2, 3]$$

                                              Type: List Integer

```
read!  ifile
```

$$[10, -10, 0, 111]$$

<div align="right">Type: List Integer</div>

The **read** operation can cause an error if one tries to read more data than is in the file. To guard against this possibility the **readIfCan** operation should be used.

```
readIfCan!  ifile
```

$$[7]$$

<div align="right">Type: Union(List Integer,...)</div>

```
readIfCan!  ifile
```

<div align="center">"failed"</div>

<div align="right">Type: Union("failed",...)</div>

You can find the current mode of the file, and the file's name.

```
iomode ifile
```

<div align="center">"input"</div>

<div align="right">Type: String</div>

```
name ifile
```

<div align="center">"/tmp/jazz1"</div>

<div align="right">Type: FileName</div>

When you are finished with a file, you should close it.

```
close!  ifile
```

<div align="center">"/tmp/jazz1"</div>

```
                              Type:  File List Integer
```

```
)system rm /tmp/jazz1
```

A limitation of the underlying LISP system is that not all values can be represented in a file. In particular, delayed values containing compiled functions cannot be saved.

For more information on related topics, see 9.81 on page 820, 9.38 on page 608, 9.41 on page 650, and 9.25 on page 558.

9.25   FileName

The `FileName` domain provides an interface to the computer's file system. Functions are provided to manipulate file names and to test properties of files.

The simplest way to use file names in the Axiom interpreter is to rely on conversion to and from strings. The syntax of these strings depends on the operating system.

```
fn:  FileName
```

```
                                        Type:  Void
```

On Linux, this is a proper file syntax:

```
fn := "/tmp/fname.input"
```

```
              "/tmp/fname.input"
```

```
                                    Type:  FileName
```

Although it is very convenient to be able to use string notation for file names in the interpreter, it is desirable to have a portable way of creating and manipulating file names from within programs.

A measure of portability is obtained by considering a file name to consist of three parts: the *directory*, the *name*, and the *extension*.

```
directory fn
```

```
              "/tmp"
```

```
                                                    Type:  String
```

```
name fn
```

```
                              "fname"
```

```
                                                    Type:  String
```

```
extension fn
```

```
                              "input"
```

```
                                                    Type:  String
```

The meaning of these three parts depends on the operating system. For example, on CMS the file "`SPADPROF INPUT M`" would have directory "`M`", name "`SPADPROF`" and extension "`INPUT`".

It is possible to create a filename from its parts.

```
fn := filename("/u/smwatt/work", "fname", "input")
```

```
                     "/u/smwatt/work/fname.input"
```

```
                                                    Type:  FileName
```

When writing programs, it is helpful to refer to directories via variables.

```
objdir := "/tmp"
```

```
                              "/tmp"
```

```
                                                    Type:  String
```

```
fn := filename(objdir, "table", "spad")
```

```
                         "/tmp/table.spad"
```

```
                                                    Type:  FileName
```

If the directory or the extension is given as an empty string, then a default is used. On AIX, the defaults are the current directory and no extension.

```
fn := filename("", "letter", "")
```

<div align="center">

"letter"

</div>

<div align="right">

Type: FileName

</div>

Three tests provide information about names in the file system.

The **exists?** operation tests whether the named file exists.

```
exists?  "/etc/passwd"
```

<div align="center">

true

</div>

<div align="right">

Type: Boolean

</div>

The operation **readable?** tells whether the named file can be read. If the file does not exist, then it cannot be read.

```
readable?  "/etc/passwd"
```

<div align="center">

true

</div>

<div align="right">

Type: Boolean

</div>

```
readable?  "/etc/security/passwd"
```

<div align="center">

false

</div>

<div align="right">

Type: Boolean

</div>

```
readable?  "/ect/passwd"
```

<div align="center">

false

</div>

<div align="right">

Type: Boolean

</div>

Likewise, the operation **writable?** tells whether the named file can be written. If the file does not exist, the test is determined by the properties of the directory.

```
writable?  "/etc/passwd"
```

```
                              false
```

```
                                              Type:  Boolean
```

```
writable?  "/dev/null"
```

```
                              true
```

```
                                              Type:  Boolean
```

```
writable?  "/etc/DoesNotExist"
```

```
                              false
```

```
                                              Type:  Boolean
```

```
writable?  "/tmp/DoesNotExist"
```

```
                              true
```

```
                                              Type:  Boolean
```

The **new** operation constructs the name of a new writable file. The argument sequence is the same as for **filename**, except that the name part is actually a prefix for a constructed unique name.

The resulting file is in the specified directory with the given extension, and the same defaults are used.

```
fn := new(objdir, "xxx", "yy")
```

```
                       "/tmp/xxx82404.yy"
```

```
                                              Type:  FileName
```

9.26   FlexibleArray

The `FlexibleArray` domain constructor creates one-dimensional arrays of elements of the same type. Flexible arrays are an attempt to provide a data type that has the best features of both one-dimensional arrays (fast, random access to

elements) and lists (flexibility). They are implemented by a fixed block of storage. When necessary for expansion, a new, larger block of storage is allocated and the elements from the old storage area are copied into the new block.

Flexible arrays have available most of the operations provided by `OneDimensionalArray` (see 9.57 on page 715 and 9.85 on page 837). Since flexible arrays are also of category `ExtensibleLinearAggregate`, they have operations `concat!`, `delete!`, `insert!`, `merge!`, `remove!`, `removeDuplicates!`, and `select!`. In addition, the operations `physicalLength` and `physicalLength!` provide user-control over expansion and contraction.

A convenient way to create a flexible array is to apply the operation `flexibleArray` to a list of values.

```
flexibleArray [i for i in 1..6]
```

$$[1, 2, 3, 4, 5, 6]$$

                              Type: FlexibleArray PositiveInteger

Create a flexible array of six zeroes.

```
f :  FARRAY INT := new(6,0)
```

$$[0, 0, 0, 0, 0, 0]$$

                                 Type: FlexibleArray Integer

For $i = 1 \ldots 6$ set the i-th element to i. Display `f`.

```
for i in 1..6 repeat f.i := i; f
```

$$[1, 2, 3, 4, 5, 6]$$

                                 Type: FlexibleArray Integer

Initially, the physical length is the same as the number of elements.

```
physicalLength f
```

$$6$$

                                    Type: PositiveInteger

Add an element to the end of `f`.

```
concat!(f,11)
```

$$[1, 2, 3, 4, 5, 6, 11]$$

<div align="right">Type: FlexibleArray Integer</div>

See that its physical length has grown.

```
physicalLength f
```

<div align="center">10</div>

<div align="right">Type: PositiveInteger</div>

Make `f` grow to have room for `15` elements.

```
physicalLength!(f,15)
```

$$[1, 2, 3, 4, 5, 6, 11]$$

<div align="right">Type: FlexibleArray Integer</div>

Concatenate the elements of `f` to itself. The physical length allows room for three more values at the end.

```
concat!(f,f)
```

$$[1, 2, 3, 4, 5, 6, 11, 1, 2, 3, 4, 5, 6, 11]$$

<div align="right">Type: FlexibleArray Integer</div>

Use `insert!` to add an element to the front of a flexible array.

```
insert!(22,f,1)
```

$$[22, 1, 2, 3, 4, 5, 6, 11, 1, 2, 3, 4, 5, 6, 11]$$

<div align="right">Type: FlexibleArray Integer</div>

Create a second flexible array from `f` consisting of the elements from index 10 forward.

```
g := f(10..)
```

$$[2, 3, 4, 5, 6, 11]$$

<div align="right">Type: FlexibleArray Integer</div>

Insert this array at the front of `f`.

```
insert!(g,f,1)
```

$$[2, 3, 4, 5, 6, 11, 22, 1, 2, 3, 4, 5, 6, 11, 1, 2, 3, 4, 5, 6, 11]$$

<div align="right">Type: FlexibleArray Integer</div>

Merge the flexible array `f` into `g` after sorting each in place.

```
merge!(sort!  f, sort!  g)
```

$$[1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 6, 6, 6, 6, 11, 11, 11, 11, 22]$$

<div align="right">Type: FlexibleArray Integer</div>

Remove duplicates in place.

```
removeDuplicates!  f
```

$$[1, 2, 3, 4, 5, 6, 11, 22]$$

<div align="right">Type: FlexibleArray Integer</div>

Remove all odd integers.

```
select!(i +-> even?  i,f)
```

$$[2, 4, 6, 22]$$

<div align="right">Type: FlexibleArray Integer</div>

All these operations have shrunk the physical length of `f`.

```
physicalLength f
```

8

To force Axiom not to shrink flexible arrays call the `shrinkable` operation with
the argument `false`. You must package call this operation. The previous value
is returned.

```
shrinkable(false)$FlexibleArray(Integer)
```

true

                                            Type: Boolean

9.27   Float

Axiom provides two kinds of floating point numbers. The domain `Float` (abbre-
viation `FLOAT`) implements a model of arbitrary precision floating point numbers.
The domain `DoubleFloat` (abbreviation `DFLOAT`) is intended to make available
hardware floating point arithmetic in Axiom. The actual model of floating point
that `DoubleFloat` provides is system-dependent. For example, on the IBM sys-
tem 370 Axiom uses IBM double precision which has fourteen hexadecimal digits
of precision or roughly sixteen decimal digits. Arbitrary precision floats allow
the user to specify the precision at which arithmetic operations are computed.
Although this is an attractive facility, it comes at a cost. Arbitrary-precision
floating-point arithmetic typically takes twenty to two hundred times more time
than hardware floating point.

For more information about Axiom's numeric and graphic facilities, see **??** on
page **??** in Section **??** on page **??**, 8.1 on page 345, and 9.17 on page 533.

9.27.1   Introduction to Float

Scientific notation is supported for input and output of floating point numbers.
A floating point number is written as a string of digits containing a decimal
point optionally followed by the letter "E", and then the exponent.

We begin by doing some calculations using arbitrary precision floats. The de-
fault precision is twenty decimal digits.

```
1.234
```

1.234

                                                        Type: Float

A decimal base for the exponent is assumed, so the number `1.234E2` denotes
$1.234 \cdot 10^2$.

```
1.234E2
```

                            123.4

                                                        Type: Float

The normal arithmetic operations are available for floating point numbers.

```
sqrt(1.2 + 2.3 / 3.4 ** 4.5)
```

                        1.0996972790 671286226

                                                        Type: Float

9.27.2   Conversion Functions

You can use conversion (**??** on page **??** in Section **??** on page **??**) to go back
and forth between `Integer`, `Fraction Integer` and `Float`, as appropriate.

```
i := 3 ::  Float
```

                            3.0

                                                        Type: Float

```
i ::  Integer
```

                             3

                                                        Type: Integer

```
i ::  Fraction Integer
```

                             3

                                              Type: Fraction Integer

Since you are explicitly asking for a conversion, you must take responsibility for
any loss of exactness.

```
r := 3/7 ::  Float
```

$$0.4285714285\ 7142857143$$

                                              Type: Float

```
r ::  Fraction Integer
```

$$\frac{3}{7}$$

                                              Type: Fraction Integer

This conversion cannot be performed: use **truncate** or **round** if that is what
you intend.

```
r ::  Integer
```

```
   Cannot convert from type Float to Integer for value
   0.4285714285 7142857143
```

The operations **truncate** and **round** truncate . . .

```
truncate 3.6
```

$$3.0$$

                                              Type: Float

and round to the nearest integral `Float` respectively.

```
round 3.6
```

$$4.0$$

                                              Type: Float

```
truncate(-3.6)
```

$$-3.0$$

<div align="right">Type: Float</div>

```
round(-3.6)
```

$$-4.0$$

<div align="right">Type: Float</div>

The operation **fractionPart** computes the fractional part of x, that is, x - truncate x.

```
fractionPart 3.6
```

$$0.6$$

<div align="right">Type: Float</div>

The operation **digits** allows the user to set the precision. It returns the previous value it was using.

```
digits 40
```

$$20$$

<div align="right">Type: PositiveInteger</div>

```
sqrt 0.2
```

$$0.4472135954\ 9995793928\ 1834733746\ 2552470881$$

<div align="right">Type: Float</div>

```
pi()$Float
```

$$3.1415926535\ 8979323846\ 2643383279\ 502884197$$

<div align="right">Type: Float</div>

The precision is only limited by the computer memory available. Calculations at 500 or more digits of precision are not difficult.

```
digits 500
```

$$40$$

<div align="right">Type: PositiveInteger</div>

```
pi()$Float
```

3.1415926535 8979323846 2643383279 5028841971 6939937510 5820974944
 5923078164 0628620899 8628034825 3421170679 8214808651 3282306647
 0938446095 5058223172 5359408128 4811174502 8410270193 8521105559
 6446229489 5493038196 4428810975 6659334461 2847564823 3786783165
 2712019091 4564856692 3460348610 4543266482 1339360726 0249141273
 7245870066 0631558817 4881520920 9628292540 9171536436 7892590360
 0113305305 4882046652 1384146951 9415116094 3305727036 5759591953
 0921861173 8193261179 3105118548 0744623799 6274956735 1885752724
 8912279381 830119491

<div align="right">Type: Float</div>

Reset **digits** to its default value.

```
digits 20
```

$$500$$

<div align="right">Type: PositiveInteger</div>

Numbers of type `Float` are represented as a record of two integers, namely, the mantissa and the exponent where the base of the exponent is binary. That is, the floating point number (`m,e`) represents the number $m \cdot 2^e$. A consequence of using a binary base is that decimal numbers can not, in general, be represented exactly.

9.27.3 Output Functions

A number of operations exist for specifying how numbers of type `Float` are to be displayed. By default, spaces are inserted every ten digits in the output for readability.[3]

---

[3]Note that you cannot include spaces in the input form of a floating point number, though you can use underscores.

Output spacing can be modified with the **outputSpacing** operation.   This inserts no spaces and then displays the value of x.

```
outputSpacing 0; x := sqrt 0.2
```

$$0.44721359549995793928$$

<div align="right">Type: Float</div>

Issue this to have the spaces inserted every 5 digits.

```
outputSpacing 5; x
```

$$0.44721\ 35954\ 99957\ 93928$$

<div align="right">Type: Float</div>

By default, the system displays floats in either fixed format or scientific format, depending on the magnitude of the number.

```
y := x/10**10
```

$$0.44721\ 35954\ 99957\ 93928\ \mathrm{E}\ -10$$

<div align="right">Type: Float</div>

A particular format may be requested with the operations **outputFloating** and **outputFixed**.

```
outputFloating(); x
```

$$0.44721\ 35954\ 99957\ 93928\ \mathrm{E}\ 0$$

<div align="right">Type: Float</div>

```
outputFixed(); y
```

$$0.00000\ 00000\ 44721\ 35954\ 99957\ 93928$$

<div align="right">Type: Float</div>

Additionally, you can ask for n digits to be displayed after the decimal point.

```
outputFloating 2; y
```

$$0.45 \text{ E} - 10$$

<div align="right">Type: Float</div>

```
outputFixed 2; x
```

$$0.45$$

<div align="right">Type: Float</div>

This resets the output printing to the default behavior.

```
outputGeneral()
```

<div align="right">Type: Void</div>

9.27.4   An Example: Determinant of a Hilbert Matrix

Consider the problem of computing the determinant of a `10` by `10` Hilbert matrix. The (i, j)-th entry of a Hilbert matrix is given by `1/(i+j+1)`.

First do the computation using rational numbers to obtain the exact result.

```
a:  Matrix Fraction Integer := matrix [ [1/(i+j+1) for j in 0..9]
for i in 0..9]
```

$$\begin{bmatrix}
1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} \\
\frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} \\
\frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} \\
\frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} \\
\frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} \\
\frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} \\
\frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} & \frac{1}{16} \\
\frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} & \frac{1}{16} & \frac{1}{17} \\
\frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} & \frac{1}{16} & \frac{1}{17} & \frac{1}{18} \\
\frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} & \frac{1}{16} & \frac{1}{17} & \frac{1}{18} & \frac{1}{19}
\end{bmatrix}$$

<div align="right">Type: Matrix Fraction Integer</div>

This version of **determinant** uses Gaussian elimination.

d:= determinant a

$$\frac{1}{46206893947914691316295628839036278726983680000000000}$$

<div align="right">Type: Fraction Integer</div>

d :: Float

$$0.21641\ 79226\ 43149\ 18691\ \text{E}\ -52$$

<div align="right">Type: Float</div>

Now use hardware floats. Note that a semicolon (;) is used to prevent the display of the matrix.

b: Matrix DoubleFloat := matrix [[1/(i+j+1$DoubleFloat) for j
in 0..9] for i in 0..9];

<div align="right">Type: Matrix DoubleFloat</div>

The result given by hardware floats is correct only to four significant digits of precision. In the jargon of numerical analysis, the Hilbert matrix is said to be "ill-conditioned."

determinant b

$$2.1643677945721411e-53$$

<div align="right">Type: DoubleFloat</div>

Now repeat the computation at a higher precision using `Float`.

digits 40

<div align="center">20</div>

<div align="right">Type: PositiveInteger</div>

c: Matrix Float := matrix [[1/(i+j+1$Float) for j in 0..9] for
i in 0..9];

<div align="right">Type: Matrix Float</div>

```
determinant c
```

$$0.21641\ 79226\ 43149\ 18690\ 60594\ 98362\ 26174\ 36159\ \mathrm{E}\ -52$$

<div align="right">Type: Float</div>

Reset **digits** to its default value.

```
digits 20
```

<div align="center">40</div>

<div align="right">Type: PositiveInteger</div>

9.28 Fraction

The `Fraction` domain implements quotients. The elements must belong to a domain of category `IntegralDomain`: multiplication must be commutative and the product of two non-zero elements must not be zero. This allows you to make fractions of most things you would think of, but don't expect to create a fraction of two matrices! The abbreviation for `Fraction` is `FRAC`.

Use "/" to create a fraction.

```
a := 11/12
```

$$\frac{11}{12}$$

<div align="right">Type: Fraction Integer</div>

```
b := 23/24
```

$$\frac{23}{24}$$

<div align="right">Type: Fraction Integer</div>

The standard arithmetic operations are available.

```
3 - a*b**2 + a + b/a
```

$$\frac{313271}{76032}$$

<div align="right">Type: Fraction Integer</div>

Extract the numerator and denominator by using **numer** and **denom**, respectively.

```
numer(a)
```

$$11$$

<div align="right">Type: PositiveInteger</div>

```
denom(b)
```

$$24$$

<div align="right">Type: PositiveInteger</div>

Operations like **max**, **min**, **negative?**, **positive?** and **zero?** are all available if they are provided for the numerators and denominators. See 9.34 on page 588 for examples.

Don't expect a useful answer from **factor**, **gcd** or **lcm** if you apply them to fractions.

```
r := (x**2 + 2*x + 1)/(x**2 - 2*x + 1)
```

$$\frac{x^2 + 2\ x + 1}{x^2 - 2\ x + 1}$$

<div align="right">Type: Fraction Polynomial Integer</div>

Since all non-zero fractions are invertible, these operations have trivial definitions.

```
factor(r)
```

$$\frac{x^2 + 2\ x + 1}{x^2 - 2\ x + 1}$$

Use **map** to apply **factor** to the numerator and denominator, which is probably what you mean.

```
map(factor,r)
```

$$\frac{(x+1)^2}{(x-1)^2}$$

Type: Fraction Factored Polynomial Integer

Other forms of fractions are available. Use `continuedFraction` to create a continued fraction.

```
continuedFraction(7/12)
```

$$\frac{1|}{|1} + \frac{1|}{|1} + \frac{1|}{|2} + \frac{1|}{|2}$$

Type: ContinuedFraction Integer

Use `partialFraction` to create a partial fraction. See and for additional information and examples.

```
partialFraction(7,12)
```

$$1 - \frac{3}{2^2} + \frac{1}{3}$$

Type: PartialFraction Integer

Use conversion to create alternative views of fractions with objects moved in and out of the numerator and denominator.

```
g := 2/3 + 4/5*%i
```

$$\frac{2}{3} + \frac{4}{5}\,i$$

Type: Complex Fraction Integer

Conversion is discussed in detail in Section **??** on page **??**.

```
g ::  FRAC COMPLEX INT
```

$$\frac{10 + 12\ i}{15}$$

<div align="right">Type: Fraction Complex Integer</div>

9.29   FullPartialFractionExpansion

The domain `FullPartialFractionExpansion` implements factor-free conversion of quotients to full partial fractions.

Our examples will all involve quotients of univariate polynomials with rational number coefficients.

```
Fx := FRAC UP(x, FRAC INT)
```

<div align="center">Fraction UnivariatePolynomial(x,Fraction Integer)</div>

<div align="right">Type: Domain</div>

Here is a simple-looking rational function.

```
f :  Fx := 36 / (x**5-2*x**4-2*x**3+4*x**2+x-2)
```

$$\frac{36}{x^5 - 2\ x^4 - 2\ x^3 + 4\ x^2 + x - 2}$$

<div align="center">Type: Fraction UnivariatePolynomial(x,Fraction Integer)</div>

We use **fullPartialFraction** to convert it to an object of type `FullPartialFractionExpansion`.

```
g := fullPartialFraction f
```

$$\frac{4}{x - 2} - \frac{4}{x + 1} + \sum_{\%A^2 - 1 = 0} \frac{-3\ \%A - 6}{(x - \%A)^2}$$

<div align="center">Type: FullPartialFractionExpansion(Fraction
Integer,UnivariatePolynomial(x,Fraction Integer))</div>

Use a coercion to change it back into a quotient.

`g ::  Fx`

$$\frac{36}{x^5 - 2\ x^4 - 2\ x^3 + 4\ x^2 + x - 2}$$

> Type: Fraction UnivariatePolynomial(x,Fraction Integer)

Full partial fractions differentiate faster than rational functions.

`g5 := D(g, 5)`

$$-\frac{480}{(x-2)^6} + \frac{480}{(x+1)^6} + \sum_{\%A^2 - 1 = 0} \frac{2160\ \%A + 4320}{(x - \%A)^7}$$

> Type: FullPartialFractionExpansion(Fraction
> Integer,UnivariatePolynomial(x,Fraction Integer))

`f5 := D(f, 5)`

$$\frac{\left(\begin{array}{l} -544320\ x^{10} + 4354560\ x^9 - 14696640\ x^8 + 28615680\ x^7 - \\[4pt] 40085280\ x^6 + 46656000\ x^5 - 39411360\ x^4 + 18247680\ x^3 - \\[4pt] 5870880\ x^2 + 3317760\ x + 246240 \end{array}\right)}{\left(\begin{array}{l} x^{20} - 12\ x^{19} + 53\ x^{18} - 76\ x^{17} - 159\ x^{16} + 676\ x^{15} - 391\ x^{14} - \\[4pt] 1596\ x^{13} + 2527\ x^{12} + 1148\ x^{11} - 4977\ x^{10} + 1372\ x^9 + \\[4pt] 4907\ x^8 - 3444\ x^7 - 2381\ x^6 + 2924\ x^5 + 276\ x^4 - \\[4pt] 1184\ x^3 + 208\ x^2 + 192\ x - 64 \end{array}\right)}$$

> Type: Fraction UnivariatePolynomial(x,Fraction Integer)

We can check that the two forms represent the same function.

`g5::Fx - f5`

$$0$$

> Type: Fraction UnivariatePolynomial(x,Fraction Integer)

Here are some examples that are more complicated.

f :   Fx := (x**5 * (x-1)) / ((x**2 + x + 1)**2 * (x-2)**3)

$$\frac{x^6 - x^5}{x^7 - 4\ x^6 + 3\ x^5 + 9\ x^3 - 6\ x^2 - 4\ x - 8}$$

            Type: Fraction UnivariatePolynomial(x,Fraction Integer)

g := fullPartialFraction f

$$\frac{\frac{1952}{2401}}{x-2} + \frac{\frac{464}{343}}{(x-2)^2} + \frac{\frac{32}{49}}{(x-2)^3} +$$

$$\sum_{\%A^2 + \%A + 1 = 0} \frac{-\frac{179}{2401}\ \%A + \frac{135}{2401}}{x - \%A} +$$

$$\sum_{\%A^2 + \%A + 1 = 0} \frac{\frac{37}{1029}\ \%A + \frac{20}{1029}}{(x - \%A)^2}$$

                    Type: FullPartialFractionExpansion(Fraction
                Integer,UnivariatePolynomial(x,Fraction Integer))

g :: Fx - f

$$0$$

        Type: Fraction UnivariatePolynomial(x,Fraction Integer)

f :   Fx := (2*x**7-7*x**5+26*x**3+8*x) /
(x**8-5*x**6+6*x**4+4*x**2-8)

$$\frac{2\ x^7 - 7\ x^5 + 26\ x^3 + 8\ x}{x^8 - 5\ x^6 + 6\ x^4 + 4\ x^2 - 8}$$

        Type: Fraction UnivariatePolynomial(x,Fraction Integer)

g := fullPartialFraction f

$$\sum_{\%A^2 - 2 = 0} \frac{\frac{1}{2}}{x - \%A} +$$

$$\sum_{\%A^2 - 2 = 0} \frac{1}{(x - \%A)^3} +$$

$$\sum_{\%A^2 + 1 = 0} \frac{\frac{1}{2}}{x - \%A}$$

                                          Type: FullPartialFractionExpansion(Fraction
                       Integer,UnivariatePolynomial(x,Fraction Integer))

```
g ::  Fx - f
```

$$0$$

              Type: Fraction UnivariatePolynomial(x,Fraction Integer)

```
f:Fx := x**3 / (x**21 + 2*x**20 + 4*x**19 + 7*x**18 + 10*x**17 +
17*x**16 + 22*x**15 + 30*x**14 + 36*x**13 + 40*x**12 + 47*x**11 +
46*x**10 + 49*x**9 + 43*x**8 + 38*x**7 + 32*x**6 + 23*x**5 +
19*x**4 + 10*x**3 + 7*x**2 + 2*x + 1)
```

$$\frac{x^3}{\left(\begin{array}{l} x^{21} + 2\ x^{20} + 4\ x^{19} + 7\ x^{18} + 10\ x^{17} + 22\ x^{15} + 30\ x^{14} + \\[2mm] 36\ x^{13} + 40\ x^{12} + 47\ x^{11} + 46\ x^{10} + 49\ x^9 + 43\ x^8 + 38\ x^7 + \\[2mm] 32\ x^6 + 23\ x^5 + 19\ x^4 + 10\ x^3 + 7\ x^2 + 2\ x + 1 \end{array}\right)}$$

              Type: Fraction UnivariatePolynomial(x,Fraction Integer)

```
g := fullPartialFraction f
```

$$\sum {}_{\%A^2+1=0} \frac{\frac{1}{2} \ \%A}{x-\%A} + \sum {}_{\%A^2+\%A+1=0} \frac{\frac{1}{9} \ \%A-\frac{19}{27}}{x-\%A} +$$

$$\sum_{\%A^2+\%A+1=0} \frac{\frac{1}{27} \ \%A - \frac{1}{27}}{\left(x-\%A\right)^2} +$$

$$\sum_{\%A^5+\%A^2+1=0} \frac{\left(\begin{array}{c} -\frac{96556567040}{912390759099} \ \%A^4 + \frac{420961732891}{912390759099} \ \%A^3 - \\[2mm] \frac{59101056149}{912390759099} \ \%A^2 - \frac{373545875923}{912390759099} \ \%A + \\[2mm] \frac{529673492498}{912390759099} \end{array}\right)}{x-\%A} +$$

$$\sum_{\%A^5+\%A^2+1=0} \frac{\left(\begin{array}{c} -\frac{5580868}{94070601} \ \%A^4 - \frac{2024443}{94070601} \ \%A^3 + \frac{4321919}{94070601} \ \%A^2 - \\[2mm] \frac{84614}{1542141} \ \%A - \frac{5070620}{94070601} \end{array}\right)}{\left(x-\%A\right)^2} +$$

$$\sum_{\%A^5+\%A^2+1=0} \frac{\left(\begin{array}{c} \frac{1610957}{94070601} \ \%A^4 + \frac{2763014}{94070601} \ \%A^3 - \frac{2016775}{94070601} \ \%A^2 + \\[2mm] \frac{266953}{94070601} \ \%A + \frac{4529359}{94070601} \end{array}\right)}{\left(x-\%A\right)^3}$$

```
              Type:  FullPartialFractionExpansion(Fraction
         Integer,UnivariatePolynomial(x,Fraction Integer))
```

This verification takes much longer than the conversion to partial fractions.

```
g ::  Fx - f
```

$$0$$

```
        Type:  Fraction UnivariatePolynomial(x,Fraction Integer)
```

For more information, see the paper: Bronstein, M and Salvy, B. "Full Partial Fraction Decomposition of Rational Functions," *Proceedings of ISSAC'93, Kiev*, ACM Press. All see 9.61 on page 730 for standard partial fraction decompositions.

9.30   GeneralSparseTable

Sometimes when working with tables there is a natural value to use as the entry in all but a few cases. The `GeneralSparseTable` constructor can be used to

provide any table type with a default value for entries. See 9.80 on page 816 for general information about tables.

Suppose we launched a fund-raising campaign to raise fifty thousand dollars. To record the contributions, we want a table with strings as keys (for the names) and integer entries (for the amount). In a data base of cash contributions, unless someone has been explicitly entered, it is reasonable to assume they have made a zero dollar contribution.

This creates a keyed access file with default entry `0`.

```
patrons:  GeneralSparseTable(String, Integer,
KeyedAccessFile(Integer), 0) := table() ;
```

                Type: GeneralSparseTable(String,Integer,KeyedAccessFile
                                                        Integer,0)

Now `patrons` can be used just as any other table. Here we record two gifts.

```
patrons."Smith" := 10500
```

                                10500

                                        Type: PositiveInteger

```
patrons."Jones" := 22000
```

                                22000

                                        Type: PositiveInteger

Now let us look up the size of the contributions from Jones and Stingy.

```
patrons."Jones"
```

                                22000

                                        Type: PositiveInteger

```
patrons."Stingy"
```

                                  0

<div align="right">Type: NonNegativeInteger</div>

Have we met our seventy thousand dollar goal?

```
reduce(+, entries patrons)
```

<div align="center">32500</div>

<div align="right">Type: PositiveInteger</div>

So the project is cancelled and we can delete the data base:

```
)system rm -r kaf*.sdata
```

9.31   GroebnerFactorizationPackage

Solving systems of polynomial equations with the Gröbner basis algorithm can often be very time consuming because, in general, the algorithm has exponential run-time. These systems, which often come from concrete applications, frequently have symmetries which are not taken advantage of by the algorithm. However, it often happens in this case that the polynomials which occur during the Gröbner calculations are reducible. Since Axiom has an excellent polynomial factorization algorithm, it is very natural to combine the Gröbner and factorization algorithms.

`GroebnerFactorizationPackage` exports the **groebnerFactorize** operation which implements a modified Gröbner basis algorithm. In this algorithm, each polynomial that is to be put into the partial list of the basis is first factored. The remaining calculation is split into as many parts as there are irreducible factors. Call these factors $p_1, \ldots, p_n$. In the branches corresponding to $p_2, \ldots, p_n$, the factor p_1 can be divided out, and so on. This package also contains operations that allow you to specify the polynomials that are not zero on the common roots of the final Gröbner basis.

Here is an example from chemistry. In a theoretical model of the cyclohexan C_6H_{12}, the six carbon atoms each sit in the center of gravity of a tetrahedron that has two hydrogen atoms and two carbon atoms at its corners. We first normalize and set the length of each edge to 1. Hence, the distances of one fixed carbon atom to each of its immediate neighbours is 1. We will denote the distances to the other three carbon atoms by x, y and z.

A. Dress developed a theory to decide whether a set of points and distances between them can be realized in an n-dimensional space. Here, of course, we have $n = 3$.

```
mfzn :  SQMATRIX(6,DMP([x,y,z],Fraction INT)) := [ [0,1,1,1,1,1],
[1,0,1,8/3,x,8/3], [1,1,0,1,8/3,y], [1,8/3,1,0,1,8/3],
[1,x,8/3,1,0,1], [1,8/3,y,8/3,1,0] ]
```

$$
\begin{bmatrix}
0 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & \frac{8}{3} & x & \frac{8}{3} \\
1 & 1 & 0 & 1 & \frac{8}{3} & y \\
1 & \frac{8}{3} & 1 & 0 & 1 & \frac{8}{3} \\
1 & x & \frac{8}{3} & 1 & 0 & 1 \\
1 & \frac{8}{3} & y & \frac{8}{3} & 1 & 0
\end{bmatrix}
$$

```
                                                               Type:
  SquareMatrix(6,DistributedMultivariatePolynomial([x,y,z],Fraction
                                                            Integer))
```

For the cyclohexan, the distances have to satisfy this equation.

```
eq := determinant mfzn
```

$$
-x^2\,y^2 + \frac{22}{3}\,x^2\,y - \frac{25}{9}\,x^2 + \frac{22}{3}\,x\,y^2 - \frac{388}{9}\,x\,y-
$$

$$
\frac{250}{27}\,x - \frac{25}{9}\,y^2 - \frac{250}{27}\,y + \frac{14575}{81}
$$

```
        Type:  DistributedMultivariatePolynomial([x,y,z],Fraction
                                                            Integer)
```

They also must satisfy the equations given by cyclic shifts of the indeterminates.

```
groebnerFactorize [eq, eval(eq, [x,y,z], [y,z,x]), eval(eq,
[x,y,z], [z,x,y])]
```

$$\left[\left[x\,y + x\,z - \frac{22}{3}\,x + y\,z - \frac{22}{3}\,y - \frac{22}{3}\,z + \frac{121}{3},\right.\right.$$

$$x\,z^2 - \frac{22}{3}\,x\,z + \frac{25}{9}\,x + y\,z^2 - \frac{22}{3}\,y\,z + \frac{25}{9}\,y - \frac{22}{3}\,z^2 + \frac{388}{9}\,z + \frac{250}{27},$$

$$y^2\,z^2 - \frac{22}{3}\,y^2\,z + \frac{25}{9}\,y^2 - \frac{22}{3}\,y\,z^2 + \frac{388}{9}\,y\,z + \frac{250}{27}\,y +$$

$$\left.\frac{25}{9}\,z^2 + \frac{250}{27}\,z - \frac{14575}{81}\right],$$

$$\left[x + y - \frac{21994}{5625}, y^2 - \frac{21994}{5625}\,y + \frac{4427}{675}, z - \frac{463}{87}\right],$$

$$\left[x^2 - \frac{1}{2}\,x\,z - \frac{11}{2}\,x - \frac{5}{6}\,z + \frac{265}{18}, y - z, z^2 - \frac{38}{3}\,z + \frac{265}{9}\right],$$

$$\left[x - \frac{25}{9}, y - \frac{11}{3}, z - \frac{11}{3}\right],$$

$$\left[x - \frac{11}{3}, y - \frac{11}{3}, z - \frac{11}{3}\right],$$

$$\left[x + \frac{5}{3}, y + \frac{5}{3}, z + \frac{5}{3}\right],$$

$$\left.\left[x - \frac{19}{3}, y + \frac{5}{3}, z + \frac{5}{3}\right]\right]$$

```
                                        Type:  List List
        DistributedMultivariatePolynomial([x,y,z],Fraction Integer)
```

The union of the solutions of this list is the solution of our original problem. If we impose positivity conditions, we get two relevant ideals. One ideal is zero-dimensional, namely $x = y = z = 11/3$, and this determines the "boat" form of the cyclohexan. The other ideal is one-dimensional, which means that we have a solution space given by one parameter. This gives the "chair" form of the cyclohexan. The parameter describes the angle of the "back of the chair."

groebnerFactorize has an optional `Boolean`-valued second argument. When it is `true` partial results are displayed, since it may happen that the calculation does not terminate in a reasonable time. See the source code for `GroebnerFactorizationPackage` in **groebf.input** for more details about the algorithms used.

9.32 Heap

The domain `Heap(S)` implements a priority queue of objects of type `S` such that the operation `extract!` removes and returns the maximum element. The implementation represents heaps as flexible arrays (see 9.26 on page 561). The representation and algorithms give complexity of $O(\log(n))$ for insertion and extractions, and $O(n)$ for construction.

Create a heap of six elements.

```
h := heap [-4,9,11,2,7,-7]
```

$$[11, 7, 9, -4, 2, -7]$$

<div align="right">Type: Heap Integer</div>

Use `insert!` to add an element.

```
insert!(3,h)
```

$$[11, 7, 9, -4, 2, -7, 3]$$

<div align="right">Type: Heap Integer</div>

The operation `extract!` removes and returns the maximum element.

```
extract!  h
```

<div align="center">11</div>

<div align="right">Type: PositiveInteger</div>

The internal structure of `h` has been appropriately adjusted.

```
h
```

$$[9, 7, 3, -4, 2, -7]$$

<div align="right">Type: Heap Integer</div>

Now `extract!` elements repeatedly until none are left, collecting the elements in a list.

```
[extract!(h) while not empty?(h)]
```

$$[9, 7, 3, 2, -4, -7]$$

<div align="right">Type: List Integer</div>

Another way to produce the same result is by defining a `heapsort` function.

```
heapsort(x) == (empty?  x => []; cons(extract!(x),heapsort x))
```

<div align="right">Void</div>

Create another sample heap.

```
h1 := heap [17,-4,9,-11,2,7,-7]
```

$$[17, 2, 9, -11, -4, 7, -7]$$

<div align="right">Type: Heap Integer</div>

Apply `heapsort` to present elements in order.

```
heapsort h1
```

$$[17, 9, 7, 2, -4, -7, -11]$$

<div align="right">Type: List Integer</div>

9.33   HexadecimalExpansion

All rationals have repeating hexadecimal expansions. The operation **hex** returns these expansions of type `HexadecimalExpansion`. Operations to access the individual numerals of a hexadecimal expansion can be obtained by converting the value to `RadixExpansion(16)`. More examples of expansions are available in the 9.15 on page 529, 9.4 on page 468, and 9.65 on page 747.

This is a hexadecimal expansion of a rational number.

```
r := hex(22/7)
```

$$3.\overline{249}$$

<div align="right">Type: HexadecimalExpansion</div>

Arithmetic is exact.

```
r + hex(6/7)
```

$$4$$

<div align="right">Type: HexadecimalExpansion</div>

The period of the expansion can be short or long . . .

```
[hex(1/i) for i in 350..354]
```

$$\left[0.00\overline{\text{BB3EE721A54D88}}, 0.00\overline{\text{BAB6561}}, 0.00\overline{\text{BA2E8}},\right.$$

$$\left.0.00\overline{\text{B9A7862A0FF465879D5F}}, 0.00\overline{\text{B92143FA36F5E02E4850FE8DBD78}}\right]$$

<div align="right">Type: List HexadecimalExpansion</div>

or very long!

```
hex(1/1007)
```

$$0.0\overline{041149783F0BF2C7D13933192AF6980619EE345E91EC2BB9D5CC}$$
$$\overline{\text{A5C071E40926E54E8DDAE24196C0B2F8A0AAD60DBA57F5D4C8}}$$
$$\overline{\text{536262210C74F1}}$$

<div align="right">Type: HexadecimalExpansion</div>

These numbers are bona fide algebraic objects.

```
p := hex(1/4)*x**2 + hex(2/3)*x + hex(4/9)
```

$$0.4 \ x^2 + 0.\overline{\text{A}} \ x + 0.\overline{\text{71C}}$$

<div align="right">Type: Polynomial HexadecimalExpansion</div>

```
q := D(p, x)
```

$$0.8 \ x + 0.\overline{\text{A}}$$

<div align="right">Type: Polynomial HexadecimalExpansion</div>

```
g := gcd(p, q)
```

$$x + 1.\overline{5}$$

<div align="right">Type: Polynomial HexadecimalExpansion</div>

9.34   Integer

Axiom provides many operations for manipulating arbitrary precision integers. In this section we will show some of those that come from `Integer` itself plus some that are implemented in other packages. More examples of using integers are in the following sections: **??** on page **??** in section **??** on page **??** 9.36 on page 599, 9.15 on page 529, 9.4 on page 468, 9.33 on page 586, and 9.65 on page 747.

9.34.1   Basic Functions

The size of an integer in Axiom is only limited by the amount of computer storage you have available. The usual arithmetic operations are available.

```
2**(5678 - 4856 + 2 * 17)
```

48048107704350081471815409251259243912395261398716822634738556100
88084200076308293086342527091412083743074572278211496076276922026
43343568752733498024953930242542523045817764949544214392905306388
478705146745768073877141698859815495632935288783334250628775936

                                              Type: PositiveInteger

There are a number of ways of working with the sign of an integer. Let's use this `x` as an example.

```
x := -101
```

$$-101$$

                                              Type: Integer

First of all, there is the absolute value function.

```
abs(x)
```

$$101$$

                                              Type: PositiveInteger

The **sign** operation returns `-1` if its argument is negative, `0` if zero and `1` if positive.

```
sign(x)
```

$$-1$$

<div align="right">Type: Integer</div>

You can determine if an integer is negative in several other ways.

```
x < 0
```

<div align="center">true</div>

<div align="right">Type: Boolean</div>

```
x <= -1
```

<div align="center">true</div>

<div align="right">Type: Boolean</div>

```
negative?(x)
```

<div align="center">true</div>

<div align="right">Type: Boolean</div>

Similarly, you can find out if it is positive.

```
x > 0
```

<div align="center">false</div>

<div align="right">Type: Boolean</div>

```
x >= 1
```

<div align="center">false</div>

<div align="right">Type: Boolean</div>

```
positive?(x)
```

<div align="center">false</div>

<div align="right">Type: Boolean</div>

This is the recommended way of determining whether an integer is zero.

```
zero?(x)
```

<div align="center">false</div>

<div align="right">Type: Boolean</div>

Use the **zero?** operation whenever you are testing any mathematical object for equality with zero. This is usually more efficient that using = (think of matrices: it is easier to tell if a matrix is zero by just checking term by term than constructing another "zero" matrix and comparing the two matrices term by term) and also avoids the problem that = is usually used for creating equations.

This is the recommended way of determining whether an integer is equal to one.

```
one?(x)
```

<div align="center">false</div>

<div align="right">Type: Boolean</div>

This syntax is used to test equality using "=". It says that you want a `Boolean` (`true` or `false`) answer rather than an equation.

```
(x = -101)@Boolean
```

<div align="center">true</div>

<div align="right">Type: Boolean</div>

The operations **odd?** and **even?** determine whether an integer is odd or even, respectively. They each return a `Boolean` object.

```
odd?(x)
```

```
                              true
```

$$\text{Type: Boolean}$$

```
even?(x)
```

```
                              false
```

$$\text{Type: Boolean}$$

The operation **gcd** computes the greatest common divisor of two integers.

```
gcd(56788,43688)
```

$$4$$

$$\text{Type: PositiveInteger}$$

The operation **lcm** computes their least common multiple.

```
lcm(56788,43688)
```

$$620238536$$

$$\text{Type: PositiveInteger}$$

To determine the maximum of two integers, use **max**.

```
max(678,567)
```

$$678$$

$$\text{Type: PositiveInteger}$$

To determine the minimum, use **min**.

```
min(678,567)
```

$$567$$

$$\text{Type: PositiveInteger}$$

The `reduce` operation is used to extend binary operations to more than two arguments. For example, you can use `reduce` to find the maximum integer in a list or compute the least common multiple of all integers in the list.

`reduce(max,[2,45,-89,78,100,-45])`

$$100$$

Type: PositiveInteger

`reduce(min,[2,45,-89,78,100,-45])`

$$-89$$

Type: Integer

`reduce(gcd,[2,45,-89,78,100,-45])`

$$1$$

Type: PositiveInteger

`reduce(lcm,[2,45,-89,78,100,-45])`

$$1041300$$

Type: PositiveInteger

The infix operator "/" is *not* used to compute the quotient of integers. Rather, it is used to create rational numbers as described in 9.28 on page 573.

`13 / 4`

$$\frac{13}{4}$$

Type: Fraction Integer

The infix operation **quo** computes the integer quotient.

`13 quo 4`

$$3$$

<div align="right">Type: PositiveInteger</div>

The infix operation **rem** computes the integer remainder.

```
13 rem 4
```

$$1$$

<div align="right">Type: PositiveInteger</div>

One integer is evenly divisible by another if the remainder is zero. The operation **exquo** can also be used. See **??** on page **??** in Section **??** on page **??** for an example.

```
zero?(167604736446952 rem 2003644)
```

<div align="center">true</div>

<div align="right">Type: Boolean</div>

The operation **divide** returns a record of the quotient and remainder and thus is more efficient when both are needed.

```
d := divide(13,4)
```

$$[quotient = 3, remainder = 1]$$

<div align="center">Type: Record(quotient: Integer,remainder: Integer)</div>

```
d.quotient
```

$$3$$

<div align="right">Type: PositiveInteger</div>

Records are discussed in detail in Section 2.4 on page 145.

```
d.remainder
```

$$1$$

<div align="right">Type: PositiveInteger</div>

9.34.2   Primes and Factorization

Use the operation **factor** to factor integers. It returns an object of type `Factored Integer`. See 9.22 on page 546 for a discussion of the manipulation of factored objects.

`factor 102400`

$$2^{12}\ 5^2$$

<div align="right">Type: Factored Integer</div>

The operation **prime?** returns `true` or `false` depending on whether its argument is a prime.

`prime?  7`

<div align="center">true</div>

<div align="right">Type: Boolean</div>

`prime?  8`

<div align="center">false</div>

<div align="right">Type: Boolean</div>

The operation **nextPrime** returns the least prime number greater than its argument.

`nextPrime 100`

<div align="center">101</div>

<div align="right">Type: PositiveInteger</div>

The operation **prevPrime** returns the greatest prime number less than its argument.

`prevPrime 100`

<div align="center">97</div>

<div align="right">Type: PositiveInteger</div>

To compute all primes between two integers (inclusively), use the operation **primes**.

```
primes(100,175)
```

$$[173, 167, 163, 157, 151, 149, 139, 137, 131, 127, 113, 109, 107, 103, 101]$$

<div align="right">Type: List Integer</div>

You might sometimes want to see the factorization of an integer when it is considered a *Gaussian integer*. See 9.11 on page 501 for more details.

```
factor(2 ::  Complex Integer)
```

$$-i \ (1+i)^2$$

<div align="right">Type: Factored Complex Integer</div>

9.34.3   Some Number Theoretic Functions

Axiom provides several number theoretic operations for integers. More examples are in 9.36 on page 599.

The operation **fibonacci** computes the Fibonacci numbers. The algorithm has running time $O\left(\log^3(n)\right)$ for argument n.

```
[fibonacci(k) for k in 0..]
```

$$[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots]$$

<div align="right">Type: Stream Integer</div>

The operation **legendre** computes the Legendre symbol for its two integer arguments where the second one is prime. If you know the second argument to be prime, use **jacobi** instead where no check is made.

```
[legendre(i,11) for i in 0..10]
```

$$[0, 1, -1, 1, 1, 1, -1, -1, -1, 1, -1]$$

<div align="right">Type: List Integer</div>

The operation **jacobi** computes the Jacobi symbol for its two integer arguments. By convention, 0 is returned if the greatest common divisor of the numerator and denominator is not 1.

```
[jacobi(i,15) for i in 0..9]
```

$$[0, 1, 1, 0, 1, 0, 0, -1, 1, 0]$$

<div align="right">Type: List Integer</div>

The operation **eulerPhi** computes the values of Euler's ϕ-function where $\phi(n)$ equals the number of positive integers less than or equal to **n** that are relatively prime to the positive integer **n**.

```
[eulerPhi i for i in 1..]
```

$$[1, 1, 2, 2, 4, 2, 6, 4, 6, 4, \ldots]$$

<div align="right">Type: Stream Integer</div>

The operation **moebiusMu** computes the Möbius μ function.

```
[moebiusMu i for i in 1..]
```

$$[1, -1, -1, 0, -1, 1, -1, 0, 0, 1, \ldots]$$

<div align="right">Type: Stream Integer</div>

Although they have somewhat limited utility, Axiom provides Roman numerals.

```
a := roman(78)
```

<div align="center">LXXVIII</div>

<div align="right">Type: RomanNumeral</div>

```
b := roman(87)
```

<div align="center">LXXXVII</div>

<div align="right">Type: RomanNumeral</div>

```
a + b
```

<div align="center">CLXV</div>

<div align="right">Type: RomanNumeral</div>

```
a * b
```

<div align="center">MMMMMMDCCLXXXVI</div>

<div align="right">Type: RomanNumeral</div>

```
b rem a
```

<div align="center">IX</div>

<div align="right">Type: RomanNumeral</div>

9.35   IntegerLinearDependence

The elements $v_1, \ldots, v_n$ of a module M over a ring R are said to be *linearly dependent over* R if there exist $c_1, \ldots, c_n$ in R, not all 0, such that $c_1 v_1 + \ldots c_n v_n = 0$. If such c_i's exist, they form what is called a *linear dependence relation over* R for the v_i's.

The package `IntegerLinearDependence` provides functions for testing whether some elements of a module over the integers are linearly dependent over the integers, and to find the linear dependence relations, if any.

Consider the domain of two by two square matrices with integer entries.

```
M := SQMATRIX(2,INT)
```

$$SquareMatrix(2, Integer)$$

<div align="right">Type: Domain</div>

Now create three such matrices.

```
m1:  M := squareMatrix matrix [ [1, 2], [0, -1] ]
```

$$\begin{bmatrix} 1 & 2 \\ 0 & -1 \end{bmatrix}$$

<div align="right">Type: SquareMatrix(2,Integer)</div>

m2:   M := squareMatrix matrix [[2, 3], [1, -2]]

$$\begin{bmatrix} 2 & 3 \\ 1 & -2 \end{bmatrix}$$

<div align="right">Type: SquareMatrix(2,Integer)</div>

m3:   M := squareMatrix matrix [[3, 4], [2, -3]]

$$\begin{bmatrix} 3 & 4 \\ 2 & -3 \end{bmatrix}$$

<div align="right">Type: SquareMatrix(2,Integer)</div>

This tells you whether m1, m2 and m3 are linearly dependent over the integers.

linearlyDependentOverZ? vector [m1, m2, m3]

<div align="center">true</div>

<div align="right">Type: Boolean</div>

Since they are linearly dependent, you can ask for the dependence relation.

c := linearDependenceOverZ vector [m1, m2, m3]

$$[1, -2, 1]$$

<div align="right">Type: Union(Vector Integer,...)</div>

This means that the following linear combination should be 0.

c.1 * m1 + c.2 * m2 + c.3 * m3

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

<div align="right">Type: SquareMatrix(2,Integer)</div>

When a given set of elements are linearly dependent over R, this also means that at least one of them can be rewritten as a linear combination of the others with coefficients in the quotient field of R.

To express a given element in terms of other elements, use the operation **solve-LinearlyOverQ**.

```
solveLinearlyOverQ(vector [m1, m3], m2)
```

$$\left[\frac{1}{2}, \frac{1}{2}\right]$$

<div align="right">Type: Union(Vector Fraction Integer,...)</div>

9.36   IntegerNumberTheoryFunctions

The `IntegerNumberTheoryFunctions` package contains a variety of operations of interest to number theorists. Many of these operations deal with divisibility properties of integers. (Recall that an integer `a` divides an integer `b` if there is an integer `c` such that `b = a * c`.)

The operation **divisors** returns a list of the divisors of an integer.

```
div144 := divisors(144)
```

$$[1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 36, 48, 72, 144]$$

<div align="right">Type: List Integer</div>

You can now compute the number of divisors of `144` and the sum of the divisors of `144` by counting and summing the elements of the list we just created.

```
#(div144)
```

<div align="center">15</div>

<div align="right">Type: PositiveInteger</div>

```
reduce(+,div144)
```

<div align="center">403</div>

<div align="right">Type: PositiveInteger</div>

Of course, you can compute the number of divisors of an integer `n`, usually denoted `d(n)`, and the sum of the divisors of an integer `n`, usually denoted $\sigma(\mathtt{n})$, without ever listing the divisors of `n`.

In Axiom, you can simply call the operations **numberOfDivisors** and **sumOfDivisors**.

```
numberOfDivisors(144)
```

<div align="center">15</div>

<div align="right">Type: PositiveInteger</div>

```
sumOfDivisors(144)
```

<div align="center">403</div>

<div align="right">Type: PositiveInteger</div>

The key is that `d(n)` and $\sigma(\mathtt{n})$ are "multiplicative functions." This means that when `n` and `m` are relatively prime, that is, when `n` and `m` have no prime factor in common, then `d(nm) = d(n) d(m)` and $\sigma(\mathtt{nm}) = \sigma(\mathtt{n})\ \sigma(\mathtt{m})$. Note that these functions are trivial to compute when `n` is a prime power and are computed for general `n` from the prime factorization of `n`. Other examples of multiplicative functions are $\sigma_k(\mathtt{n})$, the sum of the k-th powers of the divisors of `n` and $\varphi(n)$, the number of integers between 1 and `n` which are prime to `n`. The corresponding Axiom operations are called **sumOfKthPowerDivisors** and **eulerPhi**.

An interesting function is $\mu(\mathtt{n})$, the Möbius μ function, defined as follows: $\mu(\mathtt{1}) = 1$, $\mu(\mathtt{n}) = 0$, when `n` is divisible by a square, and $\mu = (-1)^k$, when n is the product of k distinct primes. The corresponding Axiom operation is **moebiusMu**. This function occurs in the following theorem:

Theorem (Möbius Inversion Formula):
Let f(n) be a function on the positive integers and let F(n) be defined by

$$F(n) = \sum_{d|n} f(n)$$

sum of f(n) over d | n where the sum is taken over the positive divisors of `n`. Then the values of `f(n)` can be recovered from the values of `F(n)`:

$$f(n) = \sum_{d|n} \mu(n)F(\frac{n}{d})$$

where again the sum is taken over the positive divisors of `n`.

When `f(n) = 1`, then `F(n) = d(n)`. Thus, if you sum $\mu(d) \cdot d(n/d)$ over the positive divisors `d` of `n`, you should always get `1`.

```
f1(n) == reduce(+,[moebiusMu(d) * numberOfDivisors(quo(n,d)) for
d in divisors(n)])
```

                                                                    Void

```
f1(200)
```

                                    1

                                        Type: PositiveInteger

```
f1(846)
```

                                    1

                                        Type: PositiveInteger

Similarly, when `f(n) = n`, then `F(n) = `σ`(n)`. Thus, if you sum μ`(d)` $\cdot$ σ`(n/d)` over the positive divisors `d` of `n`, you should always get `n`.

```
f2(n) == reduce(+,[moebiusMu(d) * sumOfDivisors(quo(n,d)) for d
in divisors(n)])
```

                                                                    Void

```
f2(200)
```

                                  200

                                        Type: PositiveInteger

```
f2(846)
```

                                  846

<div align="right">Type: PositiveInteger</div>

The Fibonacci numbers are defined by $F(1) = F(2) = 1$ and $F(n) = F(n-1) + F(n-2)$ for $n = 3, 4, \ldots$.

The operation **fibonacci** computes the n-th Fibonacci number.

```
fibonacci(25)
```

<div align="center">75025</div>

<div align="right">Type: PositiveInteger</div>

```
[fibonacci(n) for n in 1..15]
```

$$[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]$$

<div align="right">Type: List Integer</div>

Fibonacci numbers can also be expressed as sums of binomial coefficients.

```
fib(n) == reduce(+,[binomial(n-1-k,k) for k in 0..quo(n-1,2)])
```

<div align="right">Void</div>

```
fib(25)
```

<div align="center">75025</div>

<div align="right">Type: PositiveInteger</div>

```
[fib(n) for n in 1..15]
```

$$[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]$$

<div align="right">Type: List Integer</div>

Quadratic symbols can be computed with the operations **legendre** and **jacobi**. The Legendre symbol $\left(\frac{a}{p}\right)$ is defined for integers a and p with p an odd prime number. By definition, $\left(\frac{a}{p}\right) = +1$, when a is a square (mod p), $\left(\frac{a}{p}\right) = -1$, when a is not a square (mod p), and $\left(\frac{a}{p}\right) = 0$, when a is divisible by p.

You compute $\left(\frac{a}{p}\right)$ via the command `legendre(a,p)`.

```
legendre(3,5)
```

$$-1$$

<div align="right">Type: Integer</div>

```
legendre(23,691)
```

$$-1$$

<div align="right">Type: Integer</div>

The Jacobi symbol $\left(\frac{a}{n}\right)$ is the usual extension of the Legendre symbol, where **n** is an arbitrary integer. The most important property of the Jacobi symbol is the following: if **K** is a quadratic field with discriminant **d** and quadratic character χ, then $\chi(n) = (d/n)$. Thus, you can use the Jacobi symbol to compute, say, the class numbers of imaginary quadratic fields from a standard class number formula.

This function computes the class number of the imaginary quadratic field with discriminant **d**.

```
h(d) == quo(reduce(+, [jacobi(d,k) for k in 1..quo(-d, 2)]), 2 -
jacobi(d,2))
```

<div align="right">Void</div>

```
h(-163)
```

$$1$$

<div align="right">Type: PositiveInteger</div>

```
h(-499)
```

$$3$$

<div align="right">Type: PositiveInteger</div>

```
h(-1832)
```

$$26$$

<div align="right">Type: PositiveInteger</div>

9.37   Kernel

A *kernel* is a symbolic function application (such as `sin(x+ y)`) or a symbol (such as `x`). More precisely, a non-symbol kernel over a set S is an operator applied to a given list of arguments from S. The operator has type `BasicOperator` (see 9.3 on page 464) and the kernel object is usually part of an expression object (see 9.21 on page 540).

Kernels are created implicitly for you when you create expressions.

`x ::  Expression Integer`

$$x$$

> Type: Expression Integer

You can directly create a "symbol" kernel by using the **kernel** operation.

`kernel x`

$$x$$

> Type: Kernel Expression Integer

This expression has two different kernels.

`sin(x) + cos(x)`

$$\sin(x) + \cos(x)$$

> Type: Expression Integer

The operator **kernels** returns a list of the kernels in an object of type `Expression`.

`kernels %`

$$[\sin(x), \cos(x)]$$

> Type: List Kernel Expression Integer

This expression also has two different kernels.

`sin(x)**2 + sin(x) + cos(x)`

$$\sin{(x)}^2 + \sin{(x)} + \cos{(x)}$$

<div align="right">Type: Expression Integer</div>

The `sin(x)` kernel is used twice.

`kernels %`

$$[\sin{(x)}, \cos{(x)}]$$

<div align="right">Type: List Kernel Expression Integer</div>

An expression need not contain any kernels.

`kernels(1 :: Expression Integer)`

$$[\,]$$

<div align="right">Type: List Kernel Expression Integer</div>

If one or more kernels are present, one of them is designated the *main* kernel.

`mainKernel(cos(x) + tan(x))`

$$\tan{(x)}$$

<div align="right">Type: Union(Kernel Expression Integer,...)</div>

Kernels can be nested. Use **height** to determine the nesting depth.

`height kernel x`

<div align="center">1</div>

<div align="right">Type: PositiveInteger</div>

This has height 2 because the `x` has height 1 and then we apply an operator to that.

`height mainKernel(sin x)`

<div align="center">2</div>

                                              Type: PositiveInteger

```
height mainKernel(sin cos x)
```

                                3

                                              Type: PositiveInteger

```
height mainKernel(sin cos (tan x + sin x))
```

                                4

                                              Type: PositiveInteger

Use the **operator** operation to extract the operator component of the kernel.
The operator has type `BasicOperator`.

```
operator mainKernel(sin cos (tan x + sin x))
```

                            sin

                                              Type: BasicOperator

Use the **name** operation to extract the name of the operator component of the
kernel. The name has type `Symbol`. This is really just a shortcut for a two-step
process of extracting the operator and then calling **name** on the operator.

```
name mainKernel(sin cos (tan x + sin x))
```

                            sin

                                              Type: Symbol

Axiom knows about functions such as `sin`, `cos` and so on and can make kernels
and then expressions using them. To create a kernel and expression using an
arbitrary operator, use **operator**.

Now `f` can be used to create symbolic function applications.

```
f := operator 'f
```

                            f

<div align="right">Type: BasicOperator</div>

```
e := f(x, y, 10)
```

$$f(x, y, 10)$$

<div align="right">Type: Expression Integer</div>

Use the **is?** operation to learn if the operator component of a kernel is equal to a given operator.

```
is?(e, f)
```

<div align="center">true</div>

<div align="right">Type: Boolean</div>

You can also use a symbol or a string as the second argument to **is?**.

```
is?(e, 'f)
```

<div align="center">true</div>

<div align="right">Type: Boolean</div>

Use the **argument** operation to get a list containing the argument component of a kernel.

```
argument mainKernel e
```

$$[x, y, 10]$$

<div align="right">Type: List Expression Integer</div>

Conceptually, an object of type `Expression` can be thought of a quotient of multivariate polynomials, where the "variables" are kernels. The arguments of the kernels are again expressions and so the structure recurses. See 9.21 on page 540 for examples of using kernels to take apart expression objects.

9.38   KeyedAccessFile

The domain `KeyedAccessFile(S)` provides files which can be used as associative tables. Data values are stored in these files and can be retrieved according to their keys. The keys must be strings so this type behaves very much like the `StringTable(S)` domain. The difference is that keyed access files reside in secondary storage while string tables are kept in memory. For more information on table-oriented operations, see the description of `Table`.

Before a keyed access file can be used, it must first be opened. A new file can be created by opening it for output.

```
ey:  KeyedAccessFile(Integer) := open("/tmp/editor.year",
"output")
```

                               "/tmp/editor.year"

                                           Type: KeyedAccessFile Integer

Just as for vectors, tables or lists, values are saved in a keyed access file by setting elements.

```
ey."Char" := 1986
```

                                   1986

                                           Type: PositiveInteger

```
ey."Caviness" := 1985
```

                                   1985

                                           Type: PositiveInteger

```
ey."Fitch" := 1984
```

                                   1984

                                           Type: PositiveInteger

Values are retrieved using application, in any of its syntactic forms.

```
ey."Char"
```

$$1986$$

$$\text{Type: PositiveInteger}$$

```
ey("Char")
```

$$1986$$

$$\text{Type: PositiveInteger}$$

```
ey "Char"
```

$$1986$$

$$\text{Type: PositiveInteger}$$

Attempting to retrieve a non-existent element in this way causes an error. If it is not known whether a key exists, you should use the **search** operation.

```
search("Char", ey)
```

$$1986$$

$$\text{Type: Union(Integer,...)}$$

```
search("Smith", ey)
```

$$\text{"failed"}$$

$$\text{Type: Union("failed",...)}$$

When an entry is no longer needed, it can be removed from the file.

```
remove!("Char", ey)
```

$$1986$$

$$\text{Type: Union(Integer,...)}$$

The **keys** operation returns a list of all the keys for a given file.

`keys ey`

$$\left["Fitch", "Caviness"\right]$$

<div align="right">Type: List String</div>

The # operation gives the number of entries.

`#ey`

<div align="center">2</div>

<div align="right">Type: PositiveInteger</div>

The table view of keyed access files provides safe operations. That is, if the Axiom program is terminated between file operations, the file is left in a consistent, current state. This means, however, that the operations are somewhat costly. For example, after each update the file is closed.

Here we add several more items to the file, then check its contents.

`KE := Record(key:  String, entry:  Integer)`

<div align="center">Record(key: String,entry: Integer)</div>

<div align="right">Type: Domain</div>

`reopen!(ey, "output")`

<div align="center">"/tmp/editor.year"</div>

<div align="right">Type: KeyedAccessFile Integer</div>

If many items are to be added to a file at the same time, then it is more efficient to use the **write** operation.

`write!(ey, ["van Hulzen", 1983]$KE)`

$$[key = "van\ Hulzen", entry = 1983]$$

<div align="right">Type: Record(key: String,entry: Integer)</div>

```
write!(ey, ["Calmet", 1982]$KE)
```

$$[key = \texttt{"Calmet"}, entry = 1982]$$

```
                    Type:  Record(key:  String,entry:  Integer)
```

```
write!(ey, ["Wang", 1981]$KE)
```

$$[key = \texttt{"Wang"}, entry = 1981]$$

```
                    Type:  Record(key:  String,entry:  Integer)
```

```
close!  ey
```

```
                    "/tmp/editor.year"
```

```
                            Type:  KeyedAccessFile Integer
```

The **read** operation is also available from the file view, but it returns elements in a random order. It is generally clearer and more efficient to use the **keys** operation and to extract elements by key.

```
keys ey
```

$$\left[\texttt{"Wang"}, \texttt{"Calmet"}, \texttt{"van Hulzen"}, \texttt{"Fitch"}, \texttt{"Caviness"}\right]$$

```
                            Type:  List String
```

```
members ey
```

$$[1981, 1982, 1983, 1984, 1985]$$

```
                            Type:  List Integer
```

```
)system rm -r /tmp/editor.year
```

9.39   LexTriangularPackage

The `LexTriangularPackage` package constructor provides an implementation of the *lexTriangular* algorithm (D. Lazard "Solving Zero-dimensional Algebraic Systems", J. of Symbol. Comput., 1992). This algorithm decomposes a zero-dimensional variety into zero-sets of regular triangular sets. Thus the input system must have a finite number of complex solutions. Moreover, this system needs to be a lexicographical Groebner basis.

This package takes two arguments: the coefficient-ring **R** of the polynomials, which must be a `GcdDomain` and their set of variables given by **ls** a `List Symbol`.   The type of the input polynomials must be `NewSparseMultivariatePolynomial(R,V)` where **V** is `OrderedVariableList(ls)`. The abbreviation for `LexTriangularPackage` is LEXTRIPK. The main operations are **lexTriangular** and **squareFreeLexTriangular**. The later provide decompositions by means of square-free regular triangular sets, built with the `SREGSET` constructor, whereas the former uses the `REGSET` constructor. Note that these constructors also implement another algorithm for solving algebraic systems by means of regular triangular sets; in that case no computations of Groebner bases are needed and the input system may have any dimension (i.e. it may have an infinite number of solutions).

The implementation of the *lexTriangular* algorithm provided in the `LexTriangularPackage` constructor differs from that reported in "Computations of gcd over algebraic towers of simple extensions" by M. Moreno Maza and R. Rioboo (in proceedings of AAECC11, Paris, 1995). Indeed, the **squareFreeLexTriangular** operation removes all multiplicities of the solutions (i.e. the computed solutions are pairwise different) and the **lexTriangular** operation may keep some multiplicities; this later operation runs generally faster than the former.

The interest of the *lexTriangular* algorithm is due to the following experimental remark. For some examples, a triangular decomposition of a zero-dimensional variety can be computed faster via a lexicographical Groebner basis computation than by using a direct method (like that of `SREGSET` and `REGSET`). This happens typically when the total degree of the system relies essentially on its smallest variable (like in the *Katsura* systems). When this is not the case, the direct method may give better timings (like in the *Rose* system).

Of course, the direct method can also be applied to a lexicographical Groebner basis. However, the *lexTriangular* algorithm takes advantage of the structure of this basis and avoids many unnecessary computations which are performed by the direct method.

For this purpose of solving algebraic systems with a finite number of solutions, see also the `ZeroDimensionalSolvePackage`. It allows to use both strategies (the lexTriangular algorithm and the direct method) for computing either the complex or real roots of a system.

Note that the way of understanding triangular decompositions is detailed in the example of the `RegularTriangularSet` constructor.

Since the `LEXTRIPK` package constructor is limited to zero-dimensional systems, it provides a **zeroDimensional?** operation to check whether this requirement holds. There is also a **groebner** operation to compute the lexicographical Groebner basis of a set of polynomials with type `NewSparseMultivariatePolynomial(R,V)`. The elimination ordering is that given by **ls** (the greatest variable being the first element of **ls**). This basis is computed by the *FLGM* algorithm (Faugere et al. "Efficient Computation of Zero-Dimensional Groebner Bases by Change of Ordering" , J. of Symbol. Comput., 1993) implemented in the `LinGroebnerPackage` package constructor. Once a lexicographical Groebner basis is computed, then one can call the operations **lexTriangular** and **squareFreeLexTriangular**. Note that these operations admit an optional argument to produce normalized triangular sets. There is also a **zeroSetSplit** operation which does all the job from the input system; an error is produced if this system is not zero-dimensional.

Let us illustrate the facilities of the `LEXTRIPK` constructor by a famous example, the *cyclic-6 root* system.

Define the coefficient ring.

```
R := Integer
```

$$Integer$$

                                                    Type: Domain

Define the list of variables,

```
ls :  List Symbol := [a,b,c,d,e,f]
```

$$[a, b, c, d, e, f]$$

                                                Type: List Symbol

and make it an ordered set.

```
V := OVAR(ls)
```

$$\text{OrderedVariableList } [a,b,c,d,e,f]$$

                                                    Type: Domain

Define the polynomial ring.

```
P := NSMP(R, V)
```

  NewSparseMultivariatePolynomial(Integer,OrderedVariableList [a,b,c,d,e,f])

<div align="right">Type: Domain</div>

Define the polynomials.

```
p1:  P := a*b*c*d*e*f - 1
```

$$f\ e\ d\ c\ b\ a - 1$$

<div align="right">Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[a,b,c,d,e,f])</div>

```
p2:  P := a*b*c*d*e +a*b*c*d*f +a*b*c*e*f +a*b*d*e*f +a*c*d*e*f
+b*c*d*e*f
```

$$((((e+f)\ d + f\ e)\ c + f\ e\ d)\ b + f\ e\ d\ c)\ a + f\ e\ d\ c\ b$$

<div align="right">Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[a,b,c,d,e,f])</div>

```
p3:  P := a*b*c*d + a*b*c*f + a*b*e*f + a*d*e*f + b*c*d*e +
c*d*e*f
```

$$(((d+f)\ c + f\ e)\ b + f\ e\ d)\ a + e\ d\ c\ b + f\ e\ d\ c$$

<div align="right">Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[a,b,c,d,e,f])</div>

```
p4:  P := a*b*c + a*b*f + a*e*f + b*c*d + c*d*e + d*e*f
```

$$((c+f)\ b + f\ e)\ a + d\ c\ b + e\ d\ c + f\ e\ d$$

<div align="right">Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[a,b,c,d,e,f])</div>

```
p5:  P := a*b + a*f + b*c + c*d + d*e + e*f
```

$$(b + f)\, a + c\, b + d\, c + e\, d + f\, e$$

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[a,b,c,d,e,f])

```
p6:  P := a + b + c + d + e + f
```

$$a + b + c + d + e + f$$

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[a,b,c,d,e,f])

```
lp := [p1, p2, p3, p4, p5, p6]
```

$$[f\, e\, d\, c\, b\, a - 1,$$

$$((((e + f)\, d + f\, e)\, c + f\, e\, d)\, b + f\, e\, d\, c)\, a + f\, e\, d\, c\, b,$$

$$(((d + f)\, c + f\, e)\, b + f\, e\, d)\, a + e\, d\, c\, b + f\, e\, d\, c,$$

$$((c + f)\, b + f\, e)\, a + d\, c\, b + e\, d\, c + f\, e\, d,$$

$$(b + f)\, a + c\, b + d\, c + e\, d + f\, e,$$

$$a + b + c + d + e + f]$$

Type: List
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[a,b,c,d,e,f])

Now call LEXTRIPK .

```
lextripack := LEXTRIPK(R,ls)
```

$$LexTriangularPackage(Integer, [a, b, c, d, e, f])$$

Type: Domain

Compute the lexicographical Groebner basis of the system.   This may take between 5 minutes and one hour, depending on your machine.

```
lg := groebner(lp)$lextripack
```

$$[a + b + c + d + e + f,$$

$3968379498283200 \; b^2 + 15873517993132800 \; f \; b+$
$3968379498283200 \; d^2 + 15873517993132800 \; f \; d+$
$3968379498283200 \; f^3 \; e^5 - 15873517993132800 \; f^4 \; e^4+$
$23810276989699200 \; f^5 \; e^3 + \left(206355733910726400 \; f^6+\right.$
$230166010900425600) \; e^2 + \left(-729705987316687 \; f^{43}+\right.$
$1863667496867205421 \; f^{37} + 291674853771731104461 \; f^{31}+$
$365285994691106921745 \; f^{25} + 549961185828911895 \; f^{19}-$
$365048404038768439269 \; f^{13} - 292382820431504027669 \; f^7-$
$2271898467631865497 \; f) \; e - 3988812642545399 \; f^{44}+$
$10187423878429609997 \; f^{38} + 1594377523424314053637 \; f^{32}+$
$1994739308439916238065 \; f^{26} + 1596840088052642815 \; f^{20}-$
$1993494118301162145413 \; f^{14} - 1596049742289689815053 \; f^8-$
$11488171330159667449 \; f^2,$

$(23810276989699200 \; c - 23810276989699200 \; f) \; b+$
$23810276989699200 \; c^2 + 71430830969097600 \; f \; c-$
$23810276989699200 \; d^2 - 95241107958796800 \; f \; d-$
$55557312975964800 \; f^3 \; e^5 + 174608697924460800 \; f^4 \; e^4-$
$174608697924460800 \; f^5 \; e^3 + \left(-2428648252949318400 \; f^6-\right.$
$2611193709870345600) \; e^2 + \left(8305444561289527 \; f^{43}-\right.$
$21212087151945459641 \; f^{37} - 3319815883093451385381 \; f^{31}-$
$4157691646261657136445 \; f^{25} - 6072721607510764095 \; f^{19}+$
$4154986709036460221649 \; f^{13} + 3327761311138587096749 \; f^7+$
$25885340608290841637 \; f) \; e + 45815897629010329 \; f^{44}-$
$117013765582151891207 \; f^{38} - 18313166848970865074187 \; f^{32}-$
$22909971239649297438915 \; f^{26} - 16133250761305157265 \; f^{20}+$
$22897305857636178256623 \; f^{14} + 18329944781867242497923 \; f^8+$
$130258531002020420699 \; f^2,$

$(7936758996566400\ d - 7936758996566400\ f)\ b -$
$7936758996566400\ f\ d - 7936758996566400\ f^3\ e^5 +$
$23810276989699200\ f^4\ e^4 - 23810276989699200\ f^5\ e^3 +$
$\left(-337312257354072000\ f^6 - 369059293340337600\right)\ e^2 +$
$\left(1176345388640471\ f^{43} - 3004383582891473073\ f^{37} -\right.$
$470203502707246105653\ f^{31} - 588858183402644348085\ f^{25} -$
$856939308623513535\ f^{19} + 588472674242340526377\ f^{13} +$
$\left.471313241958371103517\ f^7 + 3659742549078552381\ f\right)\ e +$
$6423170513956901\ f^{44} - 16404772137036480803\ f^{38} -$
$2567419165227528774463\ f^{32} - 3211938090825682172335\ f^{26} -$
$2330490332697587485\ f^{20} + 3210100109444754864587\ f^{14} +$
$2569858315395162617847\ f^8 + 18326089487427735751\ f^2,$

$(11905138494849600\ e - 11905138494849600\ f)\ b -$
$3968379498283200\ f^3\ e^5 + 15873517993132800\ f^4\ e^4 -$
$27778656487982400\ f^5\ e^3 + \left(-208339923659868000\ f^6 -\right.$
$\left.240086959646133600\right)\ e^2 + \left(786029984751110\ f^{43} -\right.$
$2007519008182245250\ f^{37} - 314188062908073807090\ f^{31} -$
$393423667537929575250\ f^{25} - 550329120654394950\ f^{19} +$
$393196408728889612770\ f^{13} + 314892372799176495730\ f^7 +$
$\left.2409386515146668530\ f\right)\ e + 4177638546747827\ f^{44} -$
$10669685294602576381\ f^{38} - 1669852980419949524601\ f^{32} -$
$2089077057287904170745\ f^{26} - 1569899763580278795\ f^{20} +$
$2087864026859015573349\ f^{14} + 1671496085945199577969\ f^8 +$
$11940257226216280177\ f^2,$

$\left(11905138494849600\ f^6 - 11905138494849600\right)\ b -$
$15873517993132800\ f^2\ e^5 + 39683794982832000\ f^3\ e^4 -$
$39683794982832000\ f^4\ e^3 + \left(-686529653202993600\ f^{11} -\right.$
$\left.607162063237329600\ f^5\right)\ e^2 +$
$\left(65144531306704\ f^{42} - 166381280901088652\ f^{36} -\right.$
$26033434502470283472\ f^{30} - 31696259583860650140\ f^{24} +$
$971492093167581360\ f^{18} + 3222085033691389548\ f^{12} +$
$\left.25526177666070529808\ f^6 + 138603268355749244\right)\ e +$
$167620036074811\ f^{43} - 428102417974791473\ f^{37} -$
$66997243801231679313\ f^{31} - 83426716722148750485\ f^{25} +$
$203673895369980765\ f^{19} + 83523056326010432457\ f^{13} +$
$66995789640238066937\ f^7 + 478592855549587901\ f,$

$801692827936\ c^3 + 2405078483808\ f\ c^2 -$
$2405078483808\ f^2\ c - 13752945467\ f^{45} +$
$35125117815561\ f^{39} + 5496946957826433\ f^{33} +$
$6834659447749117\ f^{27} - 44484880462461\ f^{21} -$
$6873406230093057\ f^{15} - 5450844938762633\ f^9 +$
$1216586044571\ f^3,$

$(23810276989699200\ d - 23810276989699200\ f)\ c+$
$23810276989699200\ d^2 + 71430830969097600\ f\ d+$
$7936758996566400\ f^3\ e^5 - 31747035986265600\ f^4\ e^4+$
$31747035986265600\ f^5\ e^3 + \big(404774708824886400\ f^6+$
$396837949828320000\big)\ e^2 + \big(-1247372229446701\ f^{43}+$
$3185785654596621203\ f^{37} + 498594866849974751463\ f^{31}+$
$62454254584579104 7935\ f^{25} + 931085755769682885\ f^{19}-$
$62415066358241706 3387\ f^{13} - 499881859388360475647\ f^7-$
$3926885313819527351\ f)\ e - 7026011547118141\ f^{44}+$
$17944427051950691243\ f^{38} + 2808383522593986603543\ f^{32}+$
$351362414235480753 0135\ f^{26} + 2860757006705537685\ f^{20}-$
$351135673564219073 7267\ f^{14} - 2811332494697103819887\ f^8-$
$20315011631522847311\ f^2,$

$(7936758996566400\ e - 7936758996566400\ f)\ c+$
$\big(-4418748183673\ f^{43}+$
$11285568707456559\ f^{37} + 1765998617294451019\ f^{31}+$
$2173749283622606155\ f^{25} - 55788292195402895\ f^{19}-$
$2215291421788292951\ f^{13} - 1718142665347430851\ f^7+$
$30256569458230237\ f)\ e + 4418748183673\ f^{44}-$
$11285568707456559\ f^{38} - 1765998617294451019\ f^{32}-$
$2173749283622606155\ f^{26} + 55788292195402895\ f^{20}+$
$2215291421788292951\ f^{14} + 1718142665347430851\ f^8-$
$30256569458230237\ f^2,$

$\big(72152354514240\ f^6 - 72152354514240\big)\ c+$
$40950859449\ f^{43} - 104588980990367\ f^{37}-$
$16367227395575307\ f^{31} - 20268523416527355\ f^{25}+$
$442205002259535\ f^{19} + 20576059935789063\ f^{13}+$
$15997133796970563\ f^7 - 275099892785581\ f,$

$1984189749141600\ d^3 + 5952569247424800\ f\ d^2-$
$5952569247424800\ f^2\ d - 3968379498283200\ f^4\ e^5+$
$15873517993132800\ f^5\ e^4 + 17857707742274400\ e^3+$
$\big(-148814231185620000\ f^7 - 162703559429611200\ f\big)\ e^2+$
$\big(-390000914678878\ f^{44} + 996062704593756434\ f^{38}+$
$155886323972034823914\ f^{32} + 19474595614398542133 0\ f^{26}+$
$6205077595574430\ f^{20} - 194596512653299068786\ f^{14}-$
$155796897940756922666\ f^8 - 1036375759077320978\ f^2\big)\ e-$
$374998630035991\ f^{45} + 957747106595453993\ f^{39}+$
$1498891555667648916 93\ f^{33} + 187154171443494641685\ f^{27}-$
$127129015426348065\ f^{21} - 187241533243115040417\ f^{15}-$
$149719983567976534037\ f^9 - 8366540812396480 61\ f^3,$

$(5952569247424800\ e - 5952569247424800\ f)\ d-$
$3968379498283200\ f^3\ e^5 + 9920948745708000\ f^4\ e^4-$
$3968379498283200\ f^5\ e^3 + \left(-148814231185620000\ f^6-\right.$
$150798420934761600)\ e^2 + \left(492558110242553\ f^{43}-\right.$
$1257992359608074599\ f^{37} - 196883094539368513959\ f^{31}-$
$246562115745735428055\ f^{25} - 325698701993885505\ f^{19}+$
$246417769883651808111\ f^{13} + 197327352068200652911\ f^7+$
$1523373796389332143\ f)\ e + 2679481081803026\ f^{44}-$
$6843392695421906608\ f^{38} - 1071020459642646913578\ f^{32}-$
$1339789169692041240060\ f^{26} - 852746750910750210\ f^{20}+$
$1339105101971878401312\ f^{14} + 1071900289758712984762\ f^8+$
$7555239072072727756\ f^2,$

$\left(11905138494849600\ f^6 - 11905138494849600\right)\ d-$
$7936758996566400\ f^2\ e^5 + 31747035986265600\ f^3\ e^4-$
$31747035986265600\ f^4\ e^3+$
$\left(-420648226818019200\ f^{11} - 404774708824886400\ f^5\right)\ e^2+$
$\left(15336187600889\ f^{42} - 39169739565161107\ f^{36}-\right.$
$6127176127489690827\ f^{30} - 7217708742310509615\ f^{24}+$
$538628483890722735\ f^{18} + 7506804353843507643\ f^{12}+$
$5886160769782607203\ f^6 + 63576108396535879\right)\ e+$
$71737781777066\ f^{43} - 183218856207557938\ f^{37}-$
$28672874271132276078\ f^{31} - 35625223686939812010\ f^{25}+$
$164831339634084390\ f^{19} + 35724160423073052642\ f^{13}+$
$28627022578664910622\ f^7 + 187459987029680506\ f,$

$1322793166094400\ e^6 - 3968379498283200\ f\ e^5+$
$3968379498283200\ f^2\ e^4 - 5291172664377600\ f^3\ e^3+$
$\left(-230166010900425600\ f^{10} - 226197631402142400\ f^4\right)\ e^2+$
$\left(-152375364610443885\ f^{47} + 389166626064854890415\ f^{41}+\right.$
$6090609784136055898735\ f^{35} + 7616736793460879869 7275\ f^{29}+$
$2785506678599518 1125\ f^{23} - 76144952817052723145495\ f^{17}-$
$60933629892463517546975\ f^{11} - 411415071682002547795\ f^5\right)\ e-$
$209493533143822\ f^{42} + 535045979490560586\ f^{36}+$
$8373794796497355314 6\ f^{30} + 10488950708421337157 0\ f^{24}+$
$167117997269207870\ f^{18} - 104793725781390615514\ f^{12}-$
$83842685189903180394\ f^6 - 569978796672974242,$

$\left(25438330117200\ f^6 + 25438330117200\right)\ e^3+$
$\left(76314990351600\ f^7 + 76314990351600\ f\right)\ e^2+$
$\left(-1594966552735\ f^{44} + 4073543370415745\ f^{38}+\right.$
$637527159231148925\ f^{32} + 797521176113606525\ f^{26}+$
$530440941097175\ f^{20} - 797160527306433145\ f^{14}-$
$\left.638132320196044965\ f^8 - 4510507167940725\ f^2\right)\ e-$
$6036376800443\ f^{45} + 15416903421476909\ f^{39}+$
$2412807646192304449\ f^{33} + 3017679923028013705\ f^{27}+$
$1422320037411955\ f^{21} - 3016560402417843941\ f^{15}-$
$2414249368183033161\ f^9 - 16561862361763873\ f^3,$

$\left(1387545279120\ f^{12} - 1387545279120\right)\ e^2+$
$\left(4321823003\ f^{43} - 11037922310209\ f^{37}-\right.$
$1727510711947989\ f^{31} - 2165150991154425\ f^{25}-$
$5114342560755\ f^{19} + 2162682824948601\ f^{13}+$
$\left.1732620732685741\ f^7 + 13506088516033\ f\right)\ e+$
$24177661775\ f^{44} - 61749727185325\ f^{38}-$
$9664106795754225\ f^{32} - 12090487758628245\ f^{26}-$
$8787672733575\ f^{20} + 12083693383005045\ f^{14}+$
$9672870290826025\ f^8 + 68544102808525\ f^2,$

$f^{48} - 2554\ f^{42} - 399710\ f^{36} - 499722\ f^{30}+$
$499722\ f^{18} + 399710\ f^{12} + 2554\ f^6 - 1$

$$\Big]$$

```
                                        Type:  List
        NewSparseMultivariatePolynomial(Integer,OrderedVariableList
                                                 [a,b,c,d,e,f])
```

Apply lexTriangular to compute a decomposition into regular triangular sets. This should not take more than 5 seconds.

```
lexTriangular(lg,false)$lextripack
```

$$
\begin{bmatrix}
\{f^6 + 1, e^6 - 3\ f\ e^5 + 3\ f^2\ e^4 - 4\ f^3\ e^3 + 3\ f^4\ e^2 - 3\ f^5\ e - 1, \\
3\ d + f^2\ e^5 - 4\ f^3\ e^4 + 4\ f^4\ e^3 - 2\ f^5\ e^2 - 2\ e + 2\ f, c + f, \\
3\ b + 2\ f^2\ e^5 - 5\ f^3\ e^4 + 5\ f^4\ e^3 - 10\ f^5\ e^2 - 4\ e + 7\ f, \\
a - f^2\ e^5 + 3\ f^3\ e^4 - 3\ f^4\ e^3 + 4\ f^5\ e^2 + 3\ e - 3\ f\},
\end{bmatrix}
$$

$$
\{f^6 - 1, e - f, d - f, c^2 + 4\ f\ c + f^2, (c - f)\ b - f\ c - 5\ f^2, a + b + c + 3\ f\},
$$

$$
\{f^6 - 1, e - f, d - f, c - f, b^2 + 4\ f\ b + f^2, a + b + 4\ f\},
$$

$$
\{f^6 - 1, e - f, d^2 + 4\ f\ d + f^2, (d - f)\ c - f\ d - 5\ f^2, b - f, a + c + d + 3\ f\},
$$

$$
\{f^{36} - 2554\ f^{30} - 399709\ f^{24} - 502276\ f^{18} - 399709\ f^{12} - 2554\ f^6 + 1,
$$
$$
\left(161718564\ f^{12} - 161718564\right)\ e^2 + \left(-504205\ f^{31} + 1287737951\ f^{25} +
$$
$$
201539391380\ f^{19} + 253982817368\ f^{13} + 201940704665\ f^7 + 1574134601\ f\right)\ e -
$$
$$
2818405\ f^{32} + 7198203911\ f^{26} + 1126548149060\ f^{20} +
$$
$$
1416530563364\ f^{14} + 1127377589345\ f^8 + 7988820725\ f^2,
$$
$$
\left(693772639560\ f^6 - 693772639560\right)\ d - 462515093040\ f^2\ e^5 +
$$
$$
1850060372160\ f^3\ e^4 - 1850060372160\ f^4\ e^3 + \left(-24513299931120\ f^{11} -
$$
$$
23588269745040\ f^5\right)\ e^2 + \left(-890810428\ f^{30} + 2275181044754\ f^{24} +
$$
$$
355937263869776\ f^{18} + 413736880104344\ f^{12} + 342849304487996\ f^6 +
$$
$$
3704966481878\right)\ e - 4163798003\ f^{31} + 10634395752169\ f^{25} +
$$
$$
1664161760192806\ f^{19} + 2079424391370694\ f^{13} + 1668153650635921\ f^7 +
$$
$$
10924274392693\ f, \left(12614047992\ f^6 - 12614047992\right)\ c -
$$
$$
7246825\ f^{31} + 18508536599\ f^{25} + 2896249516034\ f^{19} +
$$
$$
3581539649666\ f^{13} + 2796477571739\ f^7 - 48094301893\ f,
$$
$$
\left(693772639560\ f^6 - 693772639560\right)\ b - 925030186080\ f^2\ e^5 +
$$
$$
2312575465200\ f^3\ e^4 - 2312575465200\ f^4\ e^3 + \left(-40007555547960\ f^{11} -
$$
$$
35382404617560\ f^5\right)\ e^2 + \left(-3781280823\ f^{30} + 9657492291789\ f^{24} +
$$
$$
1511158913397906\ f^{18} + 1837290892286154\ f^{12} + 1487216006594361\ f^6 +
$$
$$
8077238712093\right)\ e - 9736390478\ f^{31} + 24866827916734\ f^{25} +
$$
$$
3891495681905296\ f^{19} + 4872556418871424\ f^{13} + 3904047887269606\ f^7 +
$$
$$
27890075838538\ f, a + b + c + d + e + f\},
$$

$$
\{f^6 - 1, e^2 + 4\ f\ e + f^2, (e - f)\ d - f\ e - 5\ f^2, c - f, b - f, a + d + e + 3\ f\}]
$$

<div align="center">Type: List RegularChain(Integer,[a,b,c,d,e,f])</div>

Note that the first set of the decomposition is normalized (all initials are integer numbers) but not the second one (normalized triangular sets are defined in the description of the `NormalizedTriangularSetCategory` constructor).

So apply now lexTriangular to produce normalized triangular sets.

```
lts := lexTriangular(lg,true)$lextripack
```

$$\Big[\big\{ f^6 + 1, e^6 - 3\ f\ e^5 + 3\ f^2\ e^4 - 4\ f^3\ e^3 + 3\ f^4\ e^2 - 3\ f^5\ e - 1,$$
$$3\ d + f^2\ e^5 - 4\ f^3\ e^4 + 4\ f^4\ e^3 - 2\ f^5\ e^2 - 2\ e + 2\ f, c + f,$$
$$3\ b + 2\ f^2\ e^5 - 5\ f^3\ e^4 + 5\ f^4\ e^3 - 10\ f^5\ e^2 - 4\ e + 7\ f,$$
$$a - f^2\ e^5 + 3\ f^3\ e^4 - 3\ f^4\ e^3 + 4\ f^5\ e^2 + 3\ e - 3\ f \big\},$$

$$\big\{ f^6 - 1, e - f, d - f, c^2 + 4\ f\ c + f^2, b + c + 4\ f, a - f \big\},$$

$$\big\{ f^6 - 1, e - f, d - f, c - f, b^2 + 4\ f\ b + f^2, a + b + 4\ f \big\},$$

$$\big\{ f^6 - 1, e - f, d^2 + 4\ f\ d + f^2, c + d + 4\ f, b - f, a - f \big\},$$

$$\big\{ f^{36} - 2554\ f^{30} - 399709\ f^{24} - 502276\ f^{18} - 399709\ f^{12} - 2554\ f^6 + 1,$$
$$1387545279120\ e^2 + \big(4321823003\ f^{31} - 11037922310209\ f^{25} -$$
$$1727506390124986\ f^{19} - 2176188913464634\ f^{13} - 1732620732685741\ f^7 -$$
$$13506088516033\ f\big)\ e + 24177661775\ f^{32} - 61749727185325\ f^{26} -$$
$$9664082618092450\ f^{20} - 12152237485813570\ f^{14} - 9672870290826025\ f^8 -$$
$$68544102808525\ f^2,$$
$$1387545279120\ d + \big(-1128983050\ f^{30} + 2883434331830\ f^{24} +$$
$$451234998755840\ f^{18} + 562426491685760\ f^{12} + 447129055314890\ f^6 -$$
$$165557857270\big)\ e - 1816935351\ f^{31} + 4640452214013\ f^{25} +$$
$$726247129626942\ f^{19} + 912871801716798\ f^{13} + 726583262666877\ f^7 +$$
$$4909358645961\ f,$$
$$1387545279120\ c + 778171189\ f^{31} - 1987468196267\ f^{25} -$$
$$310993556954378\ f^{19} - 383262822316802\ f^{13} - 300335488637543\ f^7 +$$
$$5289595037041\ f,$$
$$1387545279120\ b + \big(1128983050\ f^{30} - 2883434331830\ f^{24} -$$
$$451234998755840\ f^{18} - 562426491685760\ f^{12} - 447129055314890\ f^6 +$$
$$165557857270\big)\ e - 3283058841\ f^{31} + 8384938292463\ f^{25} +$$
$$1312252817452422\ f^{19} + 1646579934064638\ f^{13} + 1306372958656407\ f^7 +$$
$$4694680112151\ f,$$
$$1387545279120\ a + 1387545279120\ e + 4321823003\ f^{31} -$$
$$11037922310209\ f^{25} - 1727506390124986\ f^{19} - 2176188913464634\ f^{13} -$$
$$1732620732685741\ f^7 - 13506088516033\ f \big\},$$

$$\big\{ f^6 - 1, e^2 + 4\ f\ e + f^2, d + e + 4\ f, c - f, b - f, a - f \big\} \Big]$$

<div align="center">Type: List RegularChain(Integer,[a,b,c,d,e,f])</div>

We check that all initials are constant.

```
[ [init(p) for p in (ts ::  List(P))] for ts in lts]
```

$$[[1, 3, 1, 3, 1, 1], [1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1],$$
$$[1387545279120, 1387545279120, 1387545279120,$$
$$1387545279120, 1387545279120, 1],$$
$$[1, 1, 1, 1, 1, 1]]$$

```
                                        Type:  List List
        NewSparseMultivariatePolynomial(Integer,OrderedVariableList
                                                  [a,b,c,d,e,f])
```

Note that each triangular set in **lts** is a lexicographical Groebner basis. Recall that a point belongs to the variety associated with **lp** if and only if it belongs to that associated with one triangular set **ts** in **lts**.

By running the **squareFreeLexTriangular** operation, we retrieve the above decomposition.

```
squareFreeLexTriangular(lg,true)$lextripack
```

$$\Big[\big\{f^6 + 1, e^6 - 3\ f\ e^5 + 3\ f^2\ e^4 - 4\ f^3\ e^3 + 3\ f^4\ e^2 - 3\ f^5\ e - 1,$$
$$3\ d + f^2\ e^5 - 4\ f^3\ e^4 + 4\ f^4\ e^3 - 2\ f^5\ e^2 - 2\ e + 2\ f,$$
$$c + f, 3\ b + 2\ f^2\ e^5 - 5\ f^3\ e^4 + 5\ f^4\ e^3 - 10\ f^5\ e^2 - 4\ e + 7\ f,$$
$$a - f^2\ e^5 + 3\ f^3\ e^4 - 3\ f^4\ e^3 + 4\ f^5\ e^2 + 3\ e - 3\ f\big\},$$

$$\big\{f^6 - 1, e - f, d - f, c^2 + 4\ f\ c + f^2, b + c + 4\ f, a - f\big\},$$

$$\big\{f^6 - 1, e - f, d - f, c - f, b^2 + 4\ f\ b + f^2, a + b + 4\ f\big\},$$

$$\big\{f^6 - 1, e - f, d^2 + 4\ f\ d + f^2, c + d + 4\ f, b - f, a - f\big\},$$

$$\big\{f^{36} - 2554\ f^{30} - 399709\ f^{24} - 502276\ f^{18} - 399709\ f^{12} - 2554\ f^6 + 1,$$
$$1387545279120\ e^2 + \big(4321823003\ f^{31} - 11037922310209\ f^{25} -$$
$$1727506390124986\ f^{19} - 2176188913464634\ f^{13} - 1732620732685741\ f^7 -$$
$$13506088516033\ f\big)\ e + 24177661775\ f^{32} - 61749727185325\ f^{26} -$$
$$9664082618092450\ f^{20} - 12152237485813570\ f^{14} - 9672870290826025\ f^8 -$$
$$68544102808525\ f^2,$$
$$1387545279120\ d + \big(-1128983050\ f^{30} + 2883434331830\ f^{24} +$$
$$451234998755840\ f^{18} + 562426491685760\ f^{12} + 447129055314890\ f^6 -$$
$$165557857270\big)\ e - 1816935351\ f^{31} + 4640452214013\ f^{25} +$$
$$726247129626942\ f^{19} + 912871801716798\ f^{13} + 726583262666877\ f^7 +$$
$$4909358645961\ f,$$
$$1387545279120\ c + 778171189\ f^{31} - 1987468196267\ f^{25} -$$
$$310993556954378\ f^{19} - 383262822316802\ f^{13} - 300335488637543\ f^7 +$$
$$5289595037041\ f,$$
$$1387545279120\ b + \big(1128983050\ f^{30} - 2883434331830\ f^{24} -$$
$$451234998755840\ f^{18} - 562426491685760\ f^{12} - 447129055314890\ f^6 +$$
$$165557857270\big)\ e - 3283058841\ f^{31} + 8384938292463\ f^{25} +$$
$$1312252817452422\ f^{19} + 1646579934064638\ f^{13} + 1306372958656407\ f^7 +$$
$$4694680112151\ f, 1387545279120\ a + 1387545279120\ e +$$
$$4321823003\ f^{31} - 11037922310209\ f^{25} - 1727506390124986\ f^{19} -$$
$$2176188913464634\ f^{13} - 1732620732685741\ f^7 - 13506088516033\ f\big\},$$

$$\big\{f^6 - 1, e^2 + 4\ f\ e + f^2, d + e + 4\ f, c - f, b - f, a - f\big\}\Big]$$

```
                                                                   Type:  List
                  SquareFreeRegularTriangularSet(Integer,IndexedExponents
                     OrderedVariableList [a,b,c,d,e,f],OrderedVariableList
[a,b,c,d,e,f],NewSparseMultivariatePolynomial(Integer,OrderedVariableList
                                                            [a,b,c,d,e,f]))
```

Thus the solutions given by **lts** are pairwise different.

We count them as follows.

```
reduce(+,[degree(ts) for ts in lts])
```

$$156$$

```
                                              Type:  PositiveInteger
```

We can investigate the triangular decomposition **lts** by using the `ZeroDimensionalSolvePackage`.

This requires to add an extra variable (smaller than the others) as follows.

```
ls2 :  List Symbol := concat(ls,new()$Symbol)
```

$$[a, b, c, d, e, f, \%A]$$

```
                                              Type:  List Symbol
```

Then we call the package.

```
zdpack := ZDSOLVE(R,ls,ls2)
```

$$ZeroDimensionalSolvePackage(Integer, [a, b, c, d, e, f], [a, b, c, d, e, f,$$

```
                                              Type:  Domain
```

We compute a univariate representation of the variety associated with the input system as follows.

```
concat [univariateSolve(ts)$zdpack for ts in lts]
```

$$\big[\big[complexRoots = ?^4 - 13\ ?^2 + 49,$$
$$coordinates =$$
$$[7\ a + \%A^3 - 6\ \%A, 21\ b + \%A^3 + \%A,$$
$$21\ c - 2\ \%A^3 + 19\ \%A, 7\ d - \%A^3 + 6\ \%A, 21\ e - \%A^3 - \%A,$$
$$21\ f + 2\ \%A^3 - 19\ \%A]\big],$$

$\big[complexRoots = ?^4 + 11\ ?^2 + 49,$
$coordinates =$
$[35\ a + 3\ \%A^3 + 19\ \%A, 35\ b + \%A^3 + 18\ \%A, 35\ c - 2\ \%A^3 - \%A,$
$35\ d - 3\ \%A^3 - 19\ \%A, 35\ e - \%A^3 - 18\ \%A, 35\ f + 2\ \%A^3 + \%A]\big],$

$\big[complexRoots = ?^8 - 12\ ?^7 + 58\ ?^6 - 120\ ?^5 +$
$207\ ?^4 - 360\ ?^3 + 802\ ?^2 - 1332\ ? + 1369,$
$coordinates =$
$[43054532\ a + 33782\ \%A^7 - 546673\ \%A^6 + 3127348\ \%A^5 - 6927123\ \%A^4 +$
$4365212\ \%A^3 - 25086957\ \%A^2 + 39582814\ \%A - 107313172,$
$43054532\ b - 33782\ \%A^7 + 546673\ \%A^6 - 3127348\ \%A^5 +$
$6927123\ \%A^4 - 4365212\ \%A^3 + 25086957\ \%A^2 -$
$39582814\ \%A + 107313172,$
$21527266\ c - 22306\ \%A^7 + 263139\ \%A^6 - 1166076\ \%A^5 + 1821805\ \%A^4 -$
$2892788\ \%A^3 + 10322663\ \%A^2 - 9026596\ \%A + 12950740,$
$43054532\ d + 22306\ \%A^7 - 263139\ \%A^6 +$
$1166076\ \%A^5 - 1821805\ \%A^4 + 2892788\ \%A^3 -$
$10322663\ \%A^2 + 30553862\ \%A - 12950740,$
$43054532\ e - 22306\ \%A^7 + 263139\ \%A^6 -$
$1166076\ \%A^5 + 1821805\ \%A^4 - 2892788\ \%A^3 +$
$10322663\ \%A^2 - 30553862\ \%A + 12950740,$
$21527266\ f + 22306\ \%A^7 - 263139\ \%A^6 +$
$1166076\ \%A^5 - 1821805\ \%A^4 + 2892788\ \%A^3 -$
$10322663\ \%A^2 + 9026596\ \%A - 12950740]\big],$

$\big[complexRoots = ?^8 + 12\ ?^7 + 58\ ?^6 + 120\ ?^5 +$
$207\ ?^4 + 360\ ?^3 + 802\ ?^2 + 1332\ ? + 1369,$
$coordinates =$
$[43054532\ a + 33782\ \%A^7 + 546673\ \%A^6 + 3127348\ \%A^5 +$
$6927123\ \%A^4 + 4365212\ \%A^3 + 25086957\ \%A^2 + 39582814\ \%A + 107313172,$
$43054532\ b - 33782\ \%A^7 - 546673\ \%A^6 - 3127348\ \%A^5 -$
$6927123\ \%A^4 - 4365212\ \%A^3 - 25086957\ \%A^2 - 39582814\ \%A - 107313172,$
$21527266\ c - 22306\ \%A^7 - 263139\ \%A^6 - 1166076\ \%A^5 -$
$1821805\ \%A^4 - 2892788\ \%A^3 - 10322663\ \%A^2 - 9026596\ \%A - 12950740,$
$43054532\ d + 22306\ \%A^7 + 263139\ \%A^6 + 1166076\ \%A^5 +$
$1821805\ \%A^4 + 2892788\ \%A^3 + 10322663\ \%A^2 + 30553862\ \%A + 12950740,$
$43054532\ e - 22306\ \%A^7 - 263139\ \%A^6 - 1166076\ \%A^5 -$
$1821805\ \%A^4 - 2892788\ \%A^3 - 10322663\ \%A^2 - 30553862\ \%A - 12950740,$
$21527266\ f + 22306\ \%A^7 + 263139\ \%A^6 + 1166076\ \%A^5 +$
$1821805\ \%A^4 + 2892788\ \%A^3 + 10322663\ \%A^2 + 9026596\ \%A + 12950740]\big],$

$\big[complexRoots = ?^4 - ?^2 + 1,$
$coordinates =$
$[a - \%A, b + \%A^3 - \%A, c + \%A^3, d + \%A, e - \%A^3 + \%A, f - \%A^3]\big],$

$\Big[complexRoots = ?^8 + 4\ ?^6 + 12\ ?^4 + 16\ ?^2 + 4,$
$coordinates =$
$\big[4\ a - 2\ \%A^7 - 7\ \%A^5 - 20\ \%A^3 - 22\ \%A,$
$4\ b + 2\ \%A^7 + 7\ \%A^5 + 20\ \%A^3 + 22\ \%A,$
$4\ c + \%A^7 + 3\ \%A^5 + 10\ \%A^3 + 10\ \%A,$
$4\ d + \%A^7 + 3\ \%A^5 + 10\ \%A^3 + 6\ \%A,$
$4\ e - \%A^7 - 3\ \%A^5 - 10\ \%A^3 - 6\ \%A,$
$\ 4\ f - \%A^7 - 3\ \%A^5 - 10\ \%A^3 - 10\ \%A\big]\Big],$

$\Big[complexRoots = ?^4 + 6\ ?^3 + 30\ ?^2 + 36\ ? + 36,$
$coordinates =$
$\big[30\ a - \%A^3 - 5\ \%A^2 - 30\ \%A - 6,$
$6\ b + \%A^3 + 5\ \%A^2 + 24\ \%A + 6,$
$30\ c - \%A^3 - 5\ \%A^2 - 6,$
$30\ d - \%A^3 - 5\ \%A^2 - 30\ \%A - 6,$
$30\ e - \%A^3 - 5\ \%A^2 - 30\ \%A - 6,$
$\ 30\ f - \%A^3 - 5\ \%A^2 - 30\ \%A - 6\big]\Big],$

$\Big[complexRoots = ?^4 - 6\ ?^3 + 30\ ?^2 - 36\ ? + 36,$
$coordinates =$
$\big[30\ a - \%A^3 + 5\ \%A^2 - 30\ \%A + 6,$
$6\ b + \%A^3 - 5\ \%A^2 + 24\ \%A - 6,$
$30\ c - \%A^3 + 5\ \%A^2 + 6,$
$30\ d - \%A^3 + 5\ \%A^2 - 30\ \%A + 6,$
$30\ e - \%A^3 + 5\ \%A^2 - 30\ \%A + 6,$
$\ 30\ f - \%A^3 + 5\ \%A^2 - 30\ \%A + 6\big]\Big],$

$\Big[complexRoots = ?^2 + 6\ ? + 6,$
$coordinates =$
$\big[a + 1, b - \%A - 5, c + \%A + 1, d + 1, e + 1, f + 1\big]\Big],$

$\Big[complexRoots = ?^2 - 6\ ? + 6,$
$coordinates =$
$\big[a - 1, b - \%A + 5, c + \%A - 1, d - 1, e - 1, f - 1\big]\Big],$

$\Big[complexRoots = ?^4 + 6\ ?^3 + 30\ ?^2 + 36\ ? + 36,$
$coordinates =$
$\big[6\ a + \%A^3 + 5\ \%A^2 + 24\ \%A + 6,$
$30\ b - \%A^3 - 5\ \%A^2 - 6,$
$30\ c - \%A^3 - 5\ \%A^2 - 30\ \%A - 6,$
$30\ d - \%A^3 - 5\ \%A^2 - 30\ \%A - 6,$
$30\ e - \%A^3 - 5\ \%A^2 - 30\ \%A - 6,$
$\ 30\ f - \%A^3 - 5\ \%A^2 - 30\ \%A - 6\big]\Big],$

$\big[complexRoots = ?^4 - 6\ ?^3 + 30\ ?^2 - 36\ ? + 36,$
$coordinates =$
$\big[6\ a + \%A^3 - 5\ \%A^2 + 24\ \%A - 6,$
$30\ b - \%A^3 + 5\ \%A^2 + 6,$
$30\ c - \%A^3 + 5\ \%A^2 - 30\ \%A + 6,$
$30\ d - \%A^3 + 5\ \%A^2 - 30\ \%A + 6,$
$30\ e - \%A^3 + 5\ \%A^2 - 30\ \%A + 6,$
$30\ f - \%A^3 + 5\ \%A^2 - 30\ \%A + 6 \big] \big],$

$\big[complexRoots = ?^2 + 6\ ? + 6,$
$coordinates = [a - \%A - 5, b + \%A + 1, c + 1, d + 1, e + 1, f + 1]\big],$

$\big[complexRoots = ?^2 - 6\ ? + 6,$
$coordinates = [a - \%A + 5, b + \%A - 1, c - 1, d - 1, e - 1, f - 1]\big],$

$\big[complexRoots = ?^4 + 6\ ?^3 + 30\ ?^2 + 36\ ? + 36,$
$coordinates =$
$\big[30\ a - \%A^3 - 5\ \%A^2 - 30\ \%A - 6,$
$30\ b - \%A^3 - 5\ \%A^2 - 30\ \%A - 6,$
$6\ c + \%A^3 + 5\ \%A^2 + 24\ \%A + 6,$
$30\ d - \%A^3 - 5\ \%A^2 - 6,$
$30\ e - \%A^3 - 5\ \%A^2 - 30\ \%A - 6,$
$30\ f - \%A^3 - 5\ \%A^2 - 30\ \%A - 6 \big] \big],$

$\big[complexRoots = ?^4 - 6\ ?^3 + 30\ ?^2 - 36\ ? + 36,$
$coordinates =$
$\big[30\ a - \%A^3 + 5\ \%A^2 - 30\ \%A + 6,$
$30\ b - \%A^3 + 5\ \%A^2 - 30\ \%A + 6,$
$6\ c + \%A^3 - 5\ \%A^2 + 24\ \%A - 6,$
$30\ d - \%A^3 + 5\ \%A^2 + 6,$
$30\ e - \%A^3 + 5\ \%A^2 - 30\ \%A + 6,$
$30\ f - \%A^3 + 5\ \%A^2 - 30\ \%A + 6 \big] \big],$

$\big[complexRoots = ?^2 + 6\ ? + 6,$
$coordinates = [a + 1, b + 1, c - \%A - 5, d + \%A + 1, e + 1, f + 1]\big],$

$\big[complexRoots = ?^2 - 6\ ? + 6,$
$coordinates = [a - 1, b - 1, c - \%A + 5, d + \%A - 1, e - 1, f - 1]\big],$

$\big[complexRoots = ?^8 + 6\ ?^7 + 16\ ?^6 + 24\ ?^5 + 18\ ?^4 - 8\ ?^2 + 4,$
$coordinates =$
$\big[2\ a + 2\ \%A^7 + 9\ \%A^6 + 18\ \%A^5 + 19\ \%A^4 + 4\ \%A^3 - 10\ \%A^2 - 2\ \%A + 4,$
$2\ b + 2\ \%A^7 + 9\ \%A^6 + 18\ \%A^5 + 19\ \%A^4 + 4\ \%A^3 - 10\ \%A^2 - 4\ \%A + 4,$
$2\ c - \%A^7 - 4\ \%A^6 - 8\ \%A^5 - 9\ \%A^4 - 4\ \%A^3 - 2\ \%A - 4,$
$2\ d + \%A^7 + 4\ \%A^6 + 8\ \%A^5 + 9\ \%A^4 + 4\ \%A^3 + 2\ \%A + 4,$
$2\ e - 2\ \%A^7 - 9\ \%A^6 - 18\ \%A^5 - 19\ \%A^4 - 4\ \%A^3 + 10\ \%A^2 + 4\ \%A - 4,$
$2\ f - 2\ \%A^7 - 9\ \%A^6 - 18\ \%A^5 - 19\ \%A^4 - 4\ \%A^3 + 10\ \%A^2 + 2\ \%A - 4 \big] \big],$

$[complexRoots =$
$?^8 + 12\ ?^7 + 64\ ?^6 + 192\ ?^5 + 432\ ?^4 + 768\ ?^3 + 1024\ ?^2 + 768\ ? + 256,$
$coordinates =$
$[1408\ a - 19\ \%A^7 - 200\ \%A^6 - 912\ \%A^5 - 2216\ \%A^4 -$
$4544\ \%A^3 - 6784\ \%A^2 - 6976\ \%A - 1792,$
$1408\ b - 37\ \%A^7 - 408\ \%A^6 - 1952\ \%A^5 - 5024\ \%A^4 -$
$10368\ \%A^3 - 16768\ \%A^2 - 17920\ \%A - 5120,$
$1408\ c + 37\ \%A^7 + 408\ \%A^6 + 1952\ \%A^5 + 5024\ \%A^4 +$
$10368\ \%A^3 + 16768\ \%A^2 + 17920\ \%A + 5120,$
$1408\ d + 19\ \%A^7 + 200\ \%A^6 + 912\ \%A^5 + 2216\ \%A^4 +$
$4544\ \%A^3 + 6784\ \%A^2 + 6976\ \%A + 1792,$
$2\ e + \%A,$
$2\ f - \%A]]\,,$

$[complexRoots = ?^8 + 4\ ?^6 + 12\ ?^4 + 16\ ?^2 + 4,$
$coordinates =$
$[4\ a - \%A^7 - 3\ \%A^5 - 10\ \%A^3 - 6\ \%A,$
$4\ b - \%A^7 - 3\ \%A^5 - 10\ \%A^3 - 10\ \%A,$
$4\ c - 2\ \%A^7 - 7\ \%A^5 - 20\ \%A^3 - 22\ \%A,$
$4\ d + 2\ \%A^7 + 7\ \%A^5 + 20\ \%A^3 + 22\ \%A,$
$4\ e + \%A^7 + 3\ \%A^5 + 10\ \%A^3 + 10\ \%A,$
$4\ f + \%A^7 + 3\ \%A^5 + 10\ \%A^3 + 6\ \%A]]\,,$

$[complexRoots = ?^8 + 16\ ?^6 - 96\ ?^4 + 256\ ?^2 + 256,$
$coordinates =$
$[512\ a - \%A^7 - 12\ \%A^5 + 176\ \%A^3 - 448\ \%A,$
$128\ b - \%A^7 - 16\ \%A^5 + 96\ \%A^3 - 256\ \%A,$
$128\ c + \%A^7 + 16\ \%A^5 - 96\ \%A^3 + 256\ \%A,$
$512\ d + \%A^7 + 12\ \%A^5 - 176\ \%A^3 + 448\ \%A,$
$2\ e + \%A,$
$2\ f - \%A]]\,,$

$[complexRoots =$
$?^8 - 12\ ?^7 + 64\ ?^6 - 192\ ?^5 + 432\ ?^4 - 768\ ?^3 + 1024\ ?^2 - 768\ ? + 256,$
$coordinates =$
$[1408\ a - 19\ \%A^7 + 200\ \%A^6 - 912\ \%A^5 + 2216\ \%A^4 -$
$4544\ \%A^3 + 6784\ \%A^2 - 6976\ \%A + 1792,$
$1408\ b - 37\ \%A^7 + 408\ \%A^6 - 1952\ \%A^5 + 5024\ \%A^4 -$
$10368\ \%A^3 + 16768\ \%A^2 - 17920\ \%A + 5120,$
$1408\ c + 37\ \%A^7 - 408\ \%A^6 + 1952\ \%A^5 - 5024\ \%A^4 +$
$10368\ \%A^3 - 16768\ \%A^2 + 17920\ \%A - 5120,$
$1408\ d + 19\ \%A^7 - 200\ \%A^6 + 912\ \%A^5 - 2216\ \%A^4 +$
$4544\ \%A^3 - 6784\ \%A^2 + 6976\ \%A - 1792,$
$2\ e + \%A,$
$2\ f - \%A]]\,,$

$\big[complexRoots = ?^8 - 6\ ?^7 + 16\ ?^6 - 24\ ?^5 + 18\ ?^4 - 8\ ?^2 + 4,$
$coordinates =$
$\big[2\ a + 2\ \%A^7 - 9\ \%A^6 + 18\ \%A^5 - 19\ \%A^4 + 4\ \%A^3 + 10\ \%A^2 - 2\ \%A - 4,$
$2\ b + 2\ \%A^7 - 9\ \%A^6 + 18\ \%A^5 - 19\ \%A^4 + 4\ \%A^3 + 10\ \%A^2 - 4\ \%A - 4,$
$2\ c - \%A^7 + 4\ \%A^6 - 8\ \%A^5 + 9\ \%A^4 - 4\ \%A^3 - 2\ \%A + 4,$
$2\ d + \%A^7 - 4\ \%A^6 + 8\ \%A^5 - 9\ \%A^4 + 4\ \%A^3 + 2\ \%A - 4,$
$2\ e - 2\ \%A^7 + 9\ \%A^6 - 18\ \%A^5 + 19\ \%A^4 - 4\ \%A^3 - 10\ \%A^2 + 4\ \%A + 4,$
$2\ f - 2\ \%A^7 + 9\ \%A^6 - 18\ \%A^5 + 19\ \%A^4 - 4\ \%A^3 - 10\ \%A^2 + 2\ \%A + 4\big]\big],$

$\big[complexRoots = ?^4 + 12\ ?^2 + 144,$
$coordinates =$
$\big[12\ a - \%A^2 - 12, 12\ b - \%A^2 - 12, 12\ c - \%A^2 - 12,$
$12\ d - \%A^2 - 12, 6\ e + \%A^2 + 3\ \%A + 12, 6\ f + \%A^2 - 3\ \%A + 12\big]\big],$

$\big[complexRoots = ?^4 + 6\ ?^3 + 30\ ?^2 + 36\ ? + 36,$
$coordinates =$
$\big[6\ a - \%A^3 - 5\ \%A^2 - 24\ \%A - 6, 30\ b + \%A^3 + 5\ \%A^2 + 30\ \%A + 6,$
$30\ c + \%A^3 + 5\ \%A^2 + 30\ \%A + 6, 30\ d + \%A^3 + 5\ \%A^2 + 30\ \%A + 6,$
$30\ e + \%A^3 + 5\ \%A^2 + 30\ \%A + 6, 30\ f + \%A^3 + 5\ \%A^2 + 6\big]\big],$

$\big[complexRoots = ?^4 - 6\ ?^3 + 30\ ?^2 - 36\ ? + 36,$
$coordinates =$
$\big[6\ a - \%A^3 + 5\ \%A^2 - 24\ \%A + 6, 30\ b + \%A^3 - 5\ \%A^2 + 30\ \%A - 6,$
$30\ c + \%A^3 - 5\ \%A^2 + 30\ \%A - 6, 30\ d + \%A^3 - 5\ \%A^2 + 30\ \%A - 6,$
$30\ e + \%A^3 - 5\ \%A^2 + 30\ \%A - 6, 30\ f + \%A^3 - 5\ \%A^2 - 6\big]\big],$

$\big[complexRoots = ?^4 + 12\ ?^2 + 144,$
$coordinates =$
$\big[12\ a + \%A^2 + 12, 12\ b + \%A^2 + 12, 12\ c + \%A^2 + 12, 12\ d + \%A^2 + 12,$
$6\ e - \%A^2 + 3\ \%A - 12, 6\ f - \%A^2 - 3\ \%A - 12\big]\big],$

$\big[complexRoots = ?^2 - 12,$
$coordinates =$
$[a - 1, b - 1, c - 1, d - 1, 2\ e + \%A + 4, 2\ f - \%A + 4]\big],$

$\big[complexRoots = ?^2 + 6\ ? + 6,$
$coordinates =$
$[a + \%A + 5, b - 1, c - 1, d - 1, e - 1, f - \%A - 1]\big],$

$\big[complexRoots = ?^2 - 6\ ? + 6,$
$coordinates =$
$[a + \%A - 5, b + 1, c + 1, d + 1, e + 1, f - \%A + 1]\big],$

$\big[complexRoots = ?^2 - 12,$
$coordinates =$
$[a + 1, b + 1, c + 1, d + 1, 2\ e + \%A - 4, 2\ f - \%A - 4]\big],$

$\big[complexRoots = ?^4 + 6 \; ?^3 + 30 \; ?^2 + 36 \; ? + 36,$

$coordinates =$

$[30 \; a - \%A^3 - 5 \; \%A^2 - 30 \; \%A - 6, 30 \; b - \%A^3 - 5 \; \%A^2 - 30 \; \%A - 6,$
$30 \; c - \%A^3 - 5 \; \%A^2 - 30 \; \%A - 6, 6 \; d + \%A^3 + 5 \; \%A^2 + 24 \; \%A + 6,$
$30 \; e - \%A^3 - 5 \; \%A^2 - 6, 30 \; f - \%A^3 - 5 \; \%A^2 - 30 \; \%A - 6]\big],$

$\big[complexRoots = ?^4 - 6 \; ?^3 + 30 \; ?^2 - 36 \; ? + 36,$

$coordinates =$

$[30 \; a - \%A^3 + 5 \; \%A^2 - 30 \; \%A + 6, 30 \; b - \%A^3 + 5 \; \%A^2 - 30 \; \%A + 6,$
$30 \; c - \%A^3 + 5 \; \%A^2 - 30 \; \%A + 6, 6 \; d + \%A^3 - 5 \; \%A^2 + 24 \; \%A - 6,$
$30 \; e - \%A^3 + 5 \; \%A^2 + 6, 30 \; f - \%A^3 + 5 \; \%A^2 - 30 \; \%A + 6]\big],$

$\big[complexRoots = ?^2 + 6 \; ? + 6,$

$coordinates =$

$[a + 1, b + 1, c + 1, d - \%A - 5, e + \%A + 1, f + 1]\big],$

$\big[complexRoots = ?^2 - 6 \; ? + 6,$

$coordinates =$

$[a - 1, b - 1, c - 1, d - \%A + 5, e + \%A - 1, f - 1]\big]$

<div align="center">

Type: List Record(complexRoots: SparseUnivariatePolynomial
                    Integer,coordinates: List Polynomial Integer)

</div>

Since the **univariateSolve** operation may split a regular set, it returns a list. This explains the use of **concat**.

Look at the last item of the result. It consists of two parts. For any complex root **?** of the univariate polynomial in the first part, we get a tuple of univariate polynomials (in **a**, ..., **f** respectively) by replacing **%A** by **?** in the second part. Each of these tuples **t** describes a point of the variety associated with **lp** by equaling to zero the polynomials in **t**.

Note that the way of reading these univariate representations is explained also in the example illustrating the `ZeroDimensionalSolvePackage` constructor.

Now, we compute the points of the variety with real coordinates.

```
concat [realSolve(ts)$zdpack for ts in lts]
```

$[[\%B23, \%B23, \%B23, \%B27, -\%B27 - 4 \; \%B23, \%B23],$

$[\%B23, \%B23, \%B23, \%B28, -\%B28 - 4 \; \%B23, \%B23],$

$[\%B24, \%B24, \%B24, \%B25, -\%B25 - 4 \; \%B24, \%B24],$

$[\%B24, \%B24, \%B24, \%B26, -\%B26 - 4 \; \%B24, \%B24],$

$[\%B29, \%B29, \%B29, \%B29, \%B33, -\%B33 - 4 \; \%B29],$

$[\%B29, \%B29, \%B29, \%B29, \%B34, -\%B34 - 4 \; \%B29],$

$[\%B30, \%B30, \%B30, \%B30, \%B31, -\%B31 - 4 \ \%B30],$

$[\%B30, \%B30, \%B30, \%B30, \%B32, -\%B32 - 4 \ \%B30],$

$[\%B35, \%B35, \%B39, -\%B39 - 4 \ \%B35, \%B35, \%B35],$

$[\%B35, \%B35, \%B40, -\%B40 - 4 \ \%B35, \%B35, \%B35],$

$[\%B36, \%B36, \%B37, -\%B37 - 4 \ \%B36, \%B36, \%B36],$

$[\%B36, \%B36, \%B38, -\%B38 - 4 \ \%B36, \%B36, \%B36],$

$[\%B41,$
$\%B51,$

$$\frac{7865521}{6006689520} \ \%B41^{31} - \frac{6696179241}{2002229840} \ \%B41^{25} -$$

$$\frac{25769893181}{49235160} \ \%B41^{19} - \frac{1975912990729}{3003344760} \ \%B41^{13} -$$

$$\frac{1048460696489}{2002229840} \ \%B41^{7} - \frac{21252634831}{6006689520} \ \%B41,$$

$$-\frac{778171189}{1387545279120} \ \%B41^{31} + \frac{1987468196267}{1387545279120} \ \%B41^{25} +$$

$$\frac{155496778477189}{693772639560} \ \%B41^{19} + \frac{191631411158401}{693772639560} \ \%B41^{13} +$$

$$\frac{300335488637543}{1387545279120} \ \%B41^{7} - \frac{755656433863}{198220754160} \ \%B41,$$

$$\frac{1094352947}{462515093040} \ \%B41^{31} - \frac{2794979430821}{462515093040} \ \%B41^{25} -$$

$$\frac{218708802908737}{231257546520} \ \%B41^{19} - \frac{91476663003591}{77085848840} \ \%B41^{13} -$$

$$\frac{145152550961823}{154171697680} \ \%B41^{7} - \frac{1564893370717}{462515093040} \ \%B41,$$

$$-\%B51 - \frac{4321823003}{1387545279120} \ \%B41^{31} + \frac{180949546069}{22746643920} \ \%B41^{25} +$$

$$\frac{863753195062493}{693772639560} \ \%B41^{19} + \frac{1088094456732317}{693772639560} \ \%B41^{13} +$$

$$\frac{1732620732685741}{1387545279120} \ \%B41^{7} + \frac{13506088516033}{1387545279120} \ \%B41\Big],$$

$[\%B41, \%B52,$

$$\frac{7865521}{6006689520}\, \%B41^{31} - \frac{6696179241}{2002229840}\, \%B41^{25} - \frac{25769893181}{49235160}\, \%B41^{19} -$$

$$\frac{1975912990729}{3003344760}\, \%B41^{13} - \frac{1048460696489}{2002229840}\, \%B41^{7} - \frac{21252634831}{6006689520}\, \%B41,$$

$$-\frac{778171189}{1387545279120}\, \%B41^{31} + \frac{1987468196267}{1387545279120}\, \%B41^{25} +$$

$$\frac{155496778477189}{693772639560}\, \%B41^{19} + \frac{191631411158401}{693772639560}\, \%B41^{13} +$$

$$\frac{300335488637543}{1387545279120}\, \%B41^{7} - \frac{755656433863}{198220754160}\, \%B41,$$

$$\frac{1094352947}{462515093040}\, \%B41^{31} - \frac{2794979430821}{462515093040}\, \%B41^{25} -$$

$$\frac{218708802908737}{231257546520}\, \%B41^{19} - \frac{91476663003591}{77085848840}\, \%B41^{13} -$$

$$\frac{145152550961823}{154171697680}\, \%B41^{7} - \frac{1564893370717}{462515093040}\, \%B41,$$

$$-\%B52 - \frac{4321823003}{1387545279120}\, \%B41^{31} + \frac{180949546069}{22746643920}\, \%B41^{25} +$$

$$\frac{863753195062493}{693772639560}\, \%B41^{19} + \frac{1088094456732317}{693772639560}\, \%B41^{13} +$$

$$\frac{1732620732685741}{1387545279120}\, \%B41^{7} + \frac{13506088516033}{1387545279120}\, \%B41 \Bigg],$$

$[\%B42, \%B49,$

$$\frac{7865521}{6006689520}\,\%B42^{31} - \frac{6696179241}{2002229840}\,\%B42^{25} -$$

$$\frac{25769893181}{49235160}\,\%B42^{19} - \frac{1975912990729}{3003344760}\,\%B42^{13} -$$

$$\frac{1048460696489}{2002229840}\,\%B42^{7} - \frac{21252634831}{6006689520}\,\%B42,$$

$$-\frac{778171189}{1387545279120}\,\%B42^{31} + \frac{1987468196267}{1387545279120}\,\%B42^{25} +$$

$$\frac{155496778477189}{693772639560}\,\%B42^{19} + \frac{191631411158401}{693772639560}\,\%B42^{13} +$$

$$\frac{300335488637543}{1387545279120}\,\%B42^{7} - \frac{755656433863}{198220754160}\,\%B42,$$

$$\frac{1094352947}{462515093040}\,\%B42^{31} - \frac{2794979430821}{462515093040}\,\%B42^{25} -$$

$$\frac{218708802908737}{231257546520}\,\%B42^{19} - \frac{91476663003591}{77085848840}\,\%B42^{13} -$$

$$\frac{145152550961823}{154171697680}\,\%B42^{7} - \frac{1564893370717}{462515093040}\,\%B42,$$

$$-\%B49 - \frac{4321823003}{1387545279120}\,\%B42^{31} + \frac{180949546069}{22746643920}\,\%B42^{25} +$$

$$\frac{863753195062493}{693772639560}\,\%B42^{19} + \frac{1088094456732317}{693772639560}\,\%B42^{13} +$$

$$\frac{1732620732685741}{1387545279120}\,\%B42^{7} + \frac{13506088516033}{1387545279120}\,\%B42\Big],$$

$[\%B42, \%B50,$

$$\frac{7865521}{6006689520}\,\%B42^{31} - \frac{6696179241}{2002229840}\,\%B42^{25} -$$

$$\frac{25769893181}{49235160}\,\%B42^{19} - \frac{1975912990729}{3003344760}\,\%B42^{13} -$$

$$\frac{1048460696489}{2002229840}\,\%B42^{7} - \frac{21252634831}{6006689520}\,\%B42,$$

$$-\frac{778171189}{1387545279120}\,\%B42^{31} + \frac{1987468196267}{1387545279120}\,\%B42^{25} +$$

$$\frac{155496778477189}{693772639560}\,\%B42^{19} + \frac{191631411158401}{693772639560}\,\%B42^{13} +$$

$$\frac{300335488637543}{1387545279120}\,\%B42^{7} - \frac{755656433863}{198220754160}\,\%B42,$$

$$\frac{1094352947}{462515093040}\,\%B42^{31} - \frac{2794979430821}{462515093040}\,\%B42^{25} -$$

$$\frac{218708802908737}{231257546520}\,\%B42^{19} - \frac{91476663003591}{77085848840}\,\%B42^{13} -$$

$$\frac{145152550961823}{154171697680}\,\%B42^{7} - \frac{1564893370717}{462515093040}\,\%B42,$$

$$-\%B50 - \frac{4321823003}{1387545279120}\,\%B42^{31} + \frac{180949546069}{22746643920}\,\%B42^{25} +$$

$$\frac{863753195062493}{693772639560}\,\%B42^{19} + \frac{1088094456732317}{693772639560}\,\%B42^{13} +$$

$$\frac{1732620732685741}{1387545279120}\,\%B42^{7} + \frac{13506088516033}{1387545279120}\,\%B42\Big],$$

$[\%B43, \%B47,$

$$\frac{7865521}{6006689520} \%B43^{31} - \frac{6696179241}{2002229840} \%B43^{25} -$$

$$\frac{25769893181}{49235160} \%B43^{19} - \frac{1975912990729}{3003344760} \%B43^{13} -$$

$$\frac{1048460696489}{2002229840} \%B43^{7} - \frac{21252634831}{6006689520} \%B43,$$

$$-\frac{778171189}{1387545279120} \%B43^{31} + \frac{1987468196267}{1387545279120} \%B43^{25} +$$

$$\frac{155496778477189}{693772639560} \%B43^{19} + \frac{191631411158401}{693772639560} \%B43^{13} +$$

$$\frac{300335488637543}{1387545279120} \%B43^{7} - \frac{755656433863}{198220754160} \%B43,$$

$$\frac{1094352947}{462515093040} \%B43^{31} - \frac{2794979430821}{462515093040} \%B43^{25} -$$

$$\frac{218708802908737}{231257546520} \%B43^{19} - \frac{91476663003591}{77085848840} \%B43^{13} -$$

$$\frac{145152550961823}{154171697680} \%B43^{7} - \frac{1564893370717}{462515093040} \%B43,$$

$$-\%B47 - \frac{4321823003}{1387545279120} \%B43^{31} + \frac{180949546069}{22746643920} \%B43^{25} +$$

$$\frac{863753195062493}{693772639560} \%B43^{19} + \frac{1088094456732317}{693772639560} \%B43^{13} +$$

$$\frac{1732620732685741}{1387545279120} \%B43^{7} + \frac{13506088516033}{1387545279120} \%B43\Big],$$

$[\%B43, \%B48,$

$$\frac{7865521}{6006689520}\,\%B43^{31} - \frac{6696179241}{2002229840}\,\%B43^{25} -$$

$$\frac{25769893181}{49235160}\,\%B43^{19} - \frac{1975912990729}{3003344760}\,\%B43^{13} -$$

$$\frac{1048460696489}{2002229840}\,\%B43^{7} - \frac{21252634831}{6006689520}\,\%B43,$$

$$-\frac{778171189}{1387545279120}\,\%B43^{31} + \frac{1987468196267}{1387545279120}\,\%B43^{25} +$$

$$\frac{155496778477189}{693772639560}\,\%B43^{19} + \frac{191631411158401}{693772639560}\,\%B43^{13} +$$

$$\frac{300335488637543}{1387545279120}\,\%B43^{7} - \frac{755656433863}{198220754160}\,\%B43,$$

$$\frac{1094352947}{462515093040}\,\%B43^{31} - \frac{2794979430821}{462515093040}\,\%B43^{25} -$$

$$\frac{218708802908737}{231257546520}\,\%B43^{19} - \frac{91476663003591}{77085848840}\,\%B43^{13} -$$

$$\frac{145152550961823}{154171697680}\,\%B43^{7} - \frac{1564893370717}{462515093040}\,\%B43,$$

$$-\%B48 - \frac{4321823003}{1387545279120}\,\%B43^{31} + \frac{180949546069}{22746643920}\,\%B43^{25} +$$

$$\frac{863753195062493}{693772639560}\,\%B43^{19} + \frac{1088094456732317}{693772639560}\,\%B43^{13} +$$

$$\left.\frac{1732620732685741}{1387545279120}\,\%B43^{7} + \frac{13506088516033}{1387545279120}\,\%B43\right],$$

$[\%B44, \%B45,$

$$\frac{7865521}{6006689520} \%B44^{31} - \frac{6696179241}{2002229840} \%B44^{25} -$$

$$\frac{25769893181}{49235160} \%B44^{19} - \frac{1975912990729}{3003344760} \%B44^{13} -$$

$$\frac{1048460696489}{2002229840} \%B44^{7} - \frac{21252634831}{6006689520} \%B44,$$

$$-\frac{778171189}{1387545279120} \%B44^{31} + \frac{1987468196267}{1387545279120} \%B44^{25} +$$

$$\frac{155496778477189}{693772639560} \%B44^{19} + \frac{191631411158401}{693772639560} \%B44^{13} +$$

$$\frac{300335488637543}{1387545279120} \%B44^{7} - \frac{755656433863}{198220754160} \%B44,$$

$$\frac{1094352947}{462515093040} \%B44^{31} - \frac{2794979430821}{462515093040} \%B44^{25} -$$

$$\frac{218708802908737}{231257546520} \%B44^{19} - \frac{91476663003591}{77085848840} \%B44^{13} -$$

$$\frac{145152550961823}{154171697680} \%B44^{7} - \frac{1564893370717}{462515093040} \%B44,$$

$$-\%B45 - \frac{4321823003}{1387545279120} \%B44^{31} + \frac{180949546069}{22746643920} \%B44^{25} +$$

$$\frac{863753195062493}{693772639560} \%B44^{19} + \frac{1088094456732317}{693772639560} \%B44^{13} +$$

$$\frac{1732620732685741}{1387545279120} \%B44^{7} + \frac{13506088516033}{1387545279120} \%B44 \Big],$$

$[\%B44, \%B46,$

$$\frac{7865521}{6006689520}\,\%B44^{31} - \frac{6696179241}{2002229840}\,\%B44^{25} -$$

$$\frac{25769893181}{49235160}\,\%B44^{19} - \frac{1975912990729}{3003344760}\,\%B44^{13} -$$

$$\frac{1048460696489}{2002229840}\,\%B44^{7} - \frac{21252634831}{6006689520}\,\%B44,$$

$$-\frac{778171189}{1387545279120}\,\%B44^{31} + \frac{1987468196267}{1387545279120}\,\%B44^{25} +$$

$$\frac{155496778477189}{693772639560}\,\%B44^{19} + \frac{191631411158401}{693772639560}\,\%B44^{13} +$$

$$\frac{300335488637543}{1387545279120}\,\%B44^{7} - \frac{755656433863}{198220754160}\,\%B44,$$

$$\frac{1094352947}{462515093040}\,\%B44^{31} - \frac{2794979430821}{462515093040}\,\%B44^{25} -$$

$$\frac{218708802908737}{231257546520}\,\%B44^{19} - \frac{91476663003591}{77085848840}\,\%B44^{13} -$$

$$\frac{145152550961823}{154171697680}\,\%B44^{7} - \frac{1564893370717}{462515093040}\,\%B44,$$

$$-\%B46 - \frac{4321823003}{1387545279120}\,\%B44^{31} + \frac{180949546069}{22746643920}\,\%B44^{25} +$$

$$\frac{863753195062493}{693772639560}\,\%B44^{19} + \frac{1088094456732317}{693772639560}\,\%B44^{13} +$$

$$\left.\frac{1732620732685741}{1387545279120}\,\%B44^{7} + \frac{13506088516033}{1387545279120}\,\%B44\right],$$

$[\%B53, \%B57, -\%B57 - 4\,\%B53, \%B53, \%B53, \%B53],$

$[\%B53, \%B58, -\%B58 - 4\,\%B53, \%B53, \%B53, \%B53],$

$[\%B54, \%B55, -\%B55 - 4\,\%B54, \%B54, \%B54, \%B54],$

$[\%B54, \%B56, -\%B56 - 4\,\%B54, \%B54, \%B54, \%B54]$ $]$

```
               Type:   List List RealClosure Fraction Integer
```

We obtain 24 points given by lists of elements in the `RealClosure` of `Fraction` of **R**. In each list, the first value corresponds to the indeterminate **f**, the second to **e** and so on. See `ZeroDimensionalSolvePackage` to learn more about the **realSolve** operation.

9.40 LazardSetSolvingPackage

The `LazardSetSolvingPackage` package constructor solves polynomial systems by means of Lazard triangular sets. However one condition is relaxed: Regular triangular sets whose saturated ideals have positive dimension are not necessarily normalized.

The decompositions are computed in two steps. First the algorithm of Moreno Maza (implemented in the `RegularTriangularSet` domain constructor) is called. Then the resulting decompositions are converted into lists of square-free regular triangular sets and the redundant components are removed. Moreover, zero-dimensional regular triangular sets are normalized.

Note that the way of understanding triangular decompositions is detailed in the example of the `RegularTriangularSet` constructor.

The `LazardSetSolvingPackage` constructor takes six arguments. The first one, **R**, is the coefficient ring of the polynomials; it must belong to the category `GcdDomain`. The second one, **E**, is the exponent monoid of the polynomials; it must belong to the category `OrderedAbelianMonoidSup`. the third one, **V**, is the ordered set of variables; it must belong to the category `OrderedSet`. The fourth one is the polynomial ring; it must belong to the category `RecursivePolynomialCategory(R,E,V)`. The fifth one is a domain of the category `RegularTriangularSetCategory(R,E,V,P)` and the last one is a domain of the category `SquareFreeRegularTriangularSetCategory(R,E,V,P)`. The abbreviation for `LazardSetSolvingPackage` is `LAZM3PK`.

N.B. For the purpose of solving zero-dimensional algebraic systems, see also `LexTriangularPackage` and `ZeroDimensionalSolvePackage`. These packages are easier to call than `LAZM3PK`. Moreover, the `ZeroDimensionalSolvePackage` package provides operations to compute either the complex roots or the real roots.

We illustrate now the use of the `LazardSetSolvingPackage` package constructor with two examples (Butcher and Vermeer).

Define the coefficient ring.

```
R := Integer
```

$$Integer$$

                                                    Type: Domain

Define the list of variables,

```
ls : List Symbol := [b1,x,y,z,t,v,u,w]
```

$$[b1, x, y, z, t, v, u, w]$$

and make it an ordered set:

```
V := OVAR(ls)
```

OrderedVariableList [b1,x,y,z,t,v,u,w]

then define the exponent monoid.

```
E := IndexedExponents V
```

IndexedExponents OrderedVariableList [b1,x,y,z,t,v,u,w]

Define the polynomial ring.

```
P  := NSMP(R, V)
```

NewSparseMultivariatePolynomial(Integer,
OrderedVariableList[b1, x, y, z, t, v, u, w])

Let the variables be polynomial.

```
b1:  P := 'b1
```

$$b1$$

```
x:  P := 'x
```

$$x$$

                                                     Type:
      NewSparseMultivariatePolynomial(Integer,OrderedVariableList
                                                 [b1,x,y,z,t,v,u,w])

y: P := 'y

$$y$$

                                                     Type:
      NewSparseMultivariatePolynomial(Integer,OrderedVariableList
                                                 [b1,x,y,z,t,v,u,w])

z: P := 'z

$$z$$

                                                     Type:
      NewSparseMultivariatePolynomial(Integer,OrderedVariableList
                                                 [b1,x,y,z,t,v,u,w])

t: P := 't

$$t$$

                                                     Type:
      NewSparseMultivariatePolynomial(Integer,OrderedVariableList
                                                 [b1,x,y,z,t,v,u,w])

u: P := 'u

$$u$$

                                                     Type:
      NewSparseMultivariatePolynomial(Integer,OrderedVariableList
                                                 [b1,x,y,z,t,v,u,w])

v: P := 'v

$$v$$

<div align="right">

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[b1,x,y,z,t,v,u,w])

</div>

```
w:  P := 'w
```

$$w$$

<div align="right">

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[b1,x,y,z,t,v,u,w])

</div>

Now call the `RegularTriangularSet` domain constructor.

```
T := REGSET(R,E,V,P)
```

> RegularTriangularSet(Integer,
>   IndexedExponentsOrderedVariableList[b1, x, y, z, t, v, u, w],
>   OrderedVariableList[b1, x, y, z, t, v, u, w],
>   NewSparseMultivariatePolynomial(Integer,
>   OrderedVariableList[b1, x, y, z, t, v, u, w]))

<div align="right">

Type: Domain

</div>

Define a polynomial system (the Butcher example).

```
p0 := b1 + y + z - t - w
```

$$b1 + y + z - t - w$$

<div align="right">

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[b1,x,y,z,t,v,u,w])

</div>

```
p1 := 2*z*u + 2*y*v + 2*t*w - 2*w**2 - w - 1
```

$$2\ v\ y + 2\ u\ z + 2\ w\ t - 2\ w^2 - w - 1$$

<div align="right">

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[b1,x,y,z,t,v,u,w])

</div>

```
p2 := 3*z*u**2 + 3*y*v**2 - 3*t*w**2 + 3*w**3 + 3*w**2 - t + 4*w
```

$$3\ v^2\ y + 3\ u^2\ z + \left(-3\ w^2 - 1\right)\ t + 3\ w^3 + 3\ w^2 + 4\ w$$

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[b1,x,y,z,t,v,u,w])

```
p3 := 6*x*z*v - 6*t*w**2 + 6*w**3 - 3*t*w + 6*w**2 - t + 4*w
```

$$6\ v\ z\ x + \left(-6\ w^2 - 3\ w - 1\right)\ t + 6\ w^3 + 6\ w^2 + 4\ w$$

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[b1,x,y,z,t,v,u,w])

```
p4 := 4*z*u**3+ 4*y*v**3+ 4*t*w**3- 4*w**4 - 6*w**3+ 4*t*w-
10*w**2- w- 1
```

$$4\ v^3\ y + 4\ u^3\ z + \left(4\ w^3 + 4\ w\right)\ t - 4\ w^4 - 6\ w^3 - 10\ w^2 - w - 1$$

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[b1,x,y,z,t,v,u,w])

```
p5 := 8*x*z*u*v +8*t*w**3 -8*w**4 +4*t*w**2 -12*w**3 +4*t*w
-14*w**2 -3*w -1
```

$$8\ u\ v\ z\ x + \left(8\ w^3 + 4\ w^2 + 4\ w\right)\ t - 8\ w^4 - 12\ w^3 - 14\ w^2 - 3\ w - 1$$

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[b1,x,y,z,t,v,u,w])

```
p6 := 12*x*z*v**2+12*t*w**3 -12*w**4 +12*t*w**2 -18*w**3 +8*t*w
-14*w**2 -w -1
```

$$12\ v^2\ z\ x + \left(12\ w^3 + 12\ w^2 + 8\ w\right)\ t - 12\ w^4 - 18\ w^3 - 14\ w^2 - w - 1$$

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[b1,x,y,z,t,v,u,w])

```
p7 := -24*t*w**3 + 24*w**4 - 24*t*w**2 + 36*w**3 - 8*t*w +
26*w**2 + 7*w + 1
```

$$\left(-24\ w^3 - 24\ w^2 - 8\ w\right)\ t + 24\ w^4 + 36\ w^3 + 26\ w^2 + 7\ w + 1$$

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[b1,x,y,z,t,v,u,w])

```
lp := [p0, p1, p2, p3, p4, p5, p6, p7]
```

$$[b1 + y + z - t - w,$$
$$2\ v\ y + 2\ u\ z + 2\ w\ t - 2\ w^2 - w - 1,$$
$$3\ v^2\ y + 3\ u^2\ z + \left(-3\ w^2 - 1\right)\ t + 3\ w^3 + 3\ w^2 + 4\ w,$$
$$6\ v\ z\ x + \left(-6\ w^2 - 3\ w - 1\right)\ t + 6\ w^3 + 6\ w^2 + 4\ w,$$
$$4\ v^3\ y + 4\ u^3\ z + \left(4\ w^3 + 4\ w\right)\ t - 4\ w^4 - 6\ w^3 - 10\ w^2 - w - 1,$$
$$8\ u\ v\ z\ x + \left(8\ w^3 + 4\ w^2 + 4\ w\right)\ t - 8\ w^4 - 12\ w^3 - 14\ w^2 - 3\ w - 1,$$
$$12\ v^2\ z\ x + \left(12\ w^3 + 12\ w^2 + 8\ w\right)\ t - 12\ w^4 - 18\ w^3 - 14\ w^2 - w - 1,$$
$$\left(-24\ w^3 - 24\ w^2 - 8\ w\right)\ t + 24\ w^4 + 36\ w^3 + 26\ w^2 + 7\ w + 1]$$

Type: List
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[b1,x,y,z,t,v,u,w])

First of all, let us solve this system in the sense of Lazard by means of the
`REGSET` constructor:

```
lts := zeroSetSplit(lp,false)$T
```

$$[\{w + 1, u, v, t + 1, b1 + y + z + 2\}, \{w + 1, v, t + 1, z, b1 + y + 2\},$$
$$\{w + 1, t + 1, z, y, b1 + 2\}, \{w + 1, v - u, t + 1, y + z, x, b1 + 2\},$$
$$\{w + 1, u, t + 1, y, x, b1 + z + 2\},$$
$$\{144\ w^5 + 216\ w^4 + 96\ w^3 + 6\ w^2 - 11\ w - 1,$$
$$\left(12\ w^2 + 9\ w + 1\right)\ u - 72\ w^5 - 108\ w^4 - 42\ w^3 - 9\ w^2 - 3\ w,$$
$$\left(12\ w^2 + 9\ w + 1\right)\ v + 36\ w^4 + 54\ w^3 + 18\ w^2,$$
$$\left(24\ w^3 + 24\ w^2 + 8\ w\right)\ t - 24\ w^4 - 36\ w^3 - 26\ w^2 - 7\ w - 1,$$
$$\left(12\ u\ v - 12\ u^2\right)\ z + \left(12\ w\ v + 12\ w^2 + 4\right)\ t + (3\ w - 5)\ v +$$
$$36\ w^4 + 42\ w^3 + 6\ w^2 - 16\ w,$$
$$2\ v\ y + 2\ u\ z + 2\ w\ t - 2\ w^2 - w - 1,$$
$$6\ v\ z\ x + \left(-6\ w^2 - 3\ w - 1\right)\ t + 6\ w^3 + 6\ w^2 + 4\ w, b1 + y + z - t - w\}]$$

Type: List RegularTriangularSet(Integer, IndexedExponents
OrderedVariableList [b1,x,y,z,t,v,u,w], OrderedVariableList
[b1,x,y,z,t,v,u,w], NewSparseMultivariatePolynomial(
Integer,OrderedVariableList [b1,x,y,z,t,v,u,w]))

We can get the dimensions of each component of a decomposition as follows.

```
[coHeight(ts) for ts in lts]
```

$$[3, 3, 3, 2, 2, 0]$$

<div align="right">Type: List NonNegativeInteger</div>

The first five sets have a simple shape. However, the last one, which has dimension zero, can be simplified by using Lazard triangular sets.

Thus we call the `SquareFreeRegularTriangularSet` domain constructor,

```
ST := SREGSET(R,E,V,P)
```

SquareFreeRegularTriangularSet(Integer,
  IndexedExponentsOrderedVariableList[b1, x, y, z, t, v, u, w],
  OrderedVariableList[b1, x, y, z, t, v, u, w],
  NewSparseMultivariatePolynomial(Integer,
  OrderedVariableList[b1, x, y, z, t, v, u, w]))

<div align="right">Type: Domain</div>

and set the `LAZM3PK` package constructor to our situation.

```
pack := LAZM3PK(R,E,V,P,T,ST)
```

LazardSetSolvingPackage(Integer,
  IndexedExponentsOrderedVariableList[b1, x, y, z, t, v, u, w],
  OrderedVariableList[b1, x, y, z, t, v, u, w],
  NewSparseMultivariatePolynomial(Integer,
  OrderedVariableList[b1, x, y, z, t, v, u, w]),
  RegularTriangularSet(Integer,
  IndexedExponentsOrderedVariableList[b1, x, y, z, t, v, u, w],
  OrderedVariableList[b1, x, y, z, t, v, u, w],
  NewSparseMultivariatePolynomial(Integer,
  OrderedVariableList[b1, x, y, z, t, v, u, w])),
  SquareFreeRegularTriangularSet(Integer,
  IndexedExponentsOrderedVariableList[b1, x, y, z, t, v, u, w],
  OrderedVariableList[b1, x, y, z, t, v, u, w],
  NewSparseMultivariatePolynomial(Integer,
  OrderedVariableList[b1, x, y, z, t, v, u, w])))

<div align="right">Type: Domain</div>

We are ready to solve the system by means of Lazard triangular sets:

```
zeroSetSplit(lp,false)$pack
```

$$[\{w + 1, t + 1, z, y, b1 + 2\},$$
$$\{w + 1, v, t + 1, z, b1 + y + 2\},$$
$$\{w + 1, u, v, t + 1, b1 + y + z + 2\},$$
$$\{w + 1, v - u, t + 1, y + z, x, b1 + 2\},$$
$$\{w + 1, u, t + 1, y, x, b1 + z + 2\},$$
$$\{144 \ w^5 + 216 \ w^4 + 96 \ w^3 + 6 \ w^2 - 11 \ w - 1,$$
$$u - 24 \ w^4 - 36 \ w^3 - 14 \ w^2 + w + 1,$$
$$3 \ v - 48 \ w^4 - 60 \ w^3 - 10 \ w^2 + 8 \ w + 2,$$
$$t - 24 \ w^4 - 36 \ w^3 - 14 \ w^2 - w + 1, 486 \ z - 2772 \ w^4 -$$
$$4662 \ w^3 - 2055 \ w^2 + 30 \ w + 127,$$
$$2916 \ y - 22752 \ w^4 - 30312 \ w^3 - 8220 \ w^2 + 2064 \ w + 1561,$$
$$356 \ x - 3696 \ w^4 - 4536 \ w^3 - 968 \ w^2 + 822 \ w + 371,$$
$$2916 \ b1 - 30600 \ w^4 - 46692 \ w^3 - 20274 \ w^2 - 8076 \ w + 593\}]$$

```
            Type:   List SquareFreeRegularTriangularSet(Integer,
       IndexedExponents OrderedVariableList [b1,x,y,z,t,v,u,w],
                        OrderedVariableList [b1,x,y,z,t,v,u,w],
   NewSparseMultivariatePolynomial(Integer, OrderedVariableList
                                      [b1,x,y,z,t,v,u,w]))
```

We see the sixth triangular set is *nicer* now: each one of its polynomials has a constant initial.

We follow with the Vermeer example. The ordering is the usual one for this system.

Define the polynomial system.

```
f0 := (w - v) ** 2 + (u - t) ** 2 - 1
```

$$t^2 - 2 \ u \ t + v^2 - 2 \ w \ v + u^2 + w^2 - 1$$

```
                                                    Type:
   NewSparseMultivariatePolynomial(Integer,OrderedVariableList
                                      [b1,x,y,z,t,v,u,w])
```

```
f1 := t ** 2 - v ** 3
```

$$t^2 - v^3$$

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[b1,x,y,z,t,v,u,w])

f2 := 2 * t * (w - v) + 3 * v ** 2 * (u - t)

$$\left(-3\ v^2 - 2\ v + 2\ w\right) t + 3\ u\ v^2$$

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[b1,x,y,z,t,v,u,w])

f3 := (3 * z * v ** 2 - 1) * (2 * z * t - 1)

$$6\ v^2\ t\ z^2 + \left(-2\ t - 3\ v^2\right) z + 1$$

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[b1,x,y,z,t,v,u,w])

lf := [f0, f1, f2, f3]

$$\begin{aligned}
&\big[t^2 - 2\ u\ t + v^2 - 2\ w\ v + u^2 + w^2 - 1,\\
&t^2 - v^3,\\
&\left(-3\ v^2 - 2\ v + 2\ w\right) t + 3\ u\ v^2,\\
&6\ v^2\ t\ z^2 + \left(-2\ t - 3\ v^2\right) z + 1\big]
\end{aligned}$$

Type: List
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[b1,x,y,z,t,v,u,w])

First of all, let us solve this system in the sense of Kalkbrener by means of the
REGSET constructor:

zeroSetSplit(lf,true)$T

$$\begin{aligned}
&\big[\big\{729\ u^6 + \left(-1458\ w^3 + 729\ w^2 - 4158\ w - 1685\right) u^4 +\\
&\left(729\ w^6 - 1458\ w^5 - 2619\ w^4 - 4892\ w^3 - 297\ w^2 + 5814\ w + 427\right) u^2 +\\
&729\ w^8 + 216\ w^7 - 2900\ w^6 - 2376\ w^5 + 3870\ w^4 +\\
&\quad 4072\ w^3 - 1188\ w^2 - 1656\ w + 529,\\
&\left(2187\ u^4 + \left(-4374\ w^3 - 972\ w^2 - 12474\ w - 2868\right) u^2 +\right.\\
&\left.2187\ w^6 - 1944\ w^5 - 10125\ w^4 - 4800\ w^3 + 2501\ w^2 + 4968\ w - 1587\right) v +\\
&\left(1944\ w^3 - 108\ w^2\right) u^2 +\\
&972\ w^6 + 3024\ w^5 - 1080\ w^4 + 496\ w^3 + 1116\ w^2,\\
&\left(3\ v^2 + 2\ v - 2\ w\right) t - 3\ u\ v^2,\\
&\left(\left(4\ v - 4\ w\right) t - 6\ u\ v^2\right) z^2 + \left(2\ t + 3\ v^2\right) z - 1\big\}\big]
\end{aligned}$$

```
Type:  List RegularTriangularSet(Integer, IndexedExponents
OrderedVariableList [b1,x,y,z,t,v,u,w], OrderedVariableList
[b1,x,y,z,t,v,u,w], NewSparseMultivariatePolynomial(Integer,
                OrderedVariableList [b1,x,y,z,t,v,u,w]))
```

We have obtained one regular chain (i.e. regular triangular set) with dimension 1. This set is in fact a characterist set of the (radical of) of the ideal generated by the input system **lf**. Thus we have only the *generic points* of the variety associated with **lf** (for the elimination ordering given by **ls**).

So let us get now a full description of this variety.

Hence, we solve this system in the sense of Lazard by means of the `REGSET` constructor:

`zeroSetSplit(lf,false)$T`

$$\Big[\Big\{729\ u^6 + \left(-1458\ w^3 + 729\ w^2 - 4158\ w - 1685\right)\ u^4 +$$
$$\left(729\ w^6 - 1458\ w^5 - 2619\ w^4 - 4892\ w^3 - 297\ w^2 + 5814\ w + 427\right)\ u^2 +$$
$$729\ w^8 + 216\ w^7 - 2900\ w^6 - 2376\ w^5 + 3870\ w^4 + 4072\ w^3 -$$
$$1188\ w^2 - 1656\ w + 529,$$
$$\left(2187\ u^4 + \left(-4374\ w^3 - 972\ w^2 - 12474\ w - 2868\right)\ u^2 +$$
$$2187\ w^6 - 1944\ w^5 - 10125\ w^4 - 4800\ w^3 + 2501\ w^2 + 4968\ w - 1587\right)\ v +$$
$$\left(1944\ w^3 - 108\ w^2\right)\ u^2 +$$
$$972\ w^6 + 3024\ w^5 - 1080\ w^4 + 496\ w^3 + 1116\ w^2,$$
$$\left(3\ v^2 + 2\ v - 2\ w\right)\ t - 3\ u\ v^2,$$
$$\left(\left(4\ v - 4\ w\right)\ t - 6\ u\ v^2\right)\ z^2 + \left(2\ t + 3\ v^2\right)\ z - 1\Big\},$$
$$\Big\{27\ w^4 + 4\ w^3 - 54\ w^2 - 36\ w + 23,$$
$$u,$$
$$\left(12\ w + 2\right)\ v - 9\ w^2 - 2\ w + 9,$$
$$6\ t^2 - 2\ v - 3\ w^2 + 2\ w + 3,$$
$$2\ t\ z - 1\Big\},$$
$$\Big\{59049\ w^6 + 91854\ w^5 - 45198\ w^4 + 145152\ w^3 + 63549\ w^2 + 60922\ w + 21420,$$
$$\left(31484448266904\ w^5 - 18316865522574\ w^4 + 23676995746098\ w^3 + 6657857188965\ w^2 +$$
$$8904703998546\ w + 3890631403260\right)\ u^2 + 94262810316408\ w^5 - 82887296576616\ w^4 +$$
$$89801831438784\ w^3 + 28141734167208\ w^2 + 38070359425432\ w + 16003865949120,$$
$$\left(243\ w^2 + 36\ w + 85\right)\ v^2 + \left(-81\ u^2 - 162\ w^3 + 36\ w^2 + 154\ w + 72\right)\ v - 72\ w^3 + 4\ w^2,$$
$$\left(3\ v^2 + 2\ v - 2\ w\right)\ t - 3\ u\ v^2,$$
$$\left(\left(4\ v - 4\ w\right)\ t - 6\ u\ v^2\right)\ z^2 + \left(2\ t + 3\ v^2\right)\ z - 1\Big\},$$
$$\Big\{27\ w^4 + 4\ w^3 - 54\ w^2 - 36\ w + 23, u,$$
$$\left(12\ w + 2\right)\ v - 9\ w^2 - 2\ w + 9,$$
$$6\ t^2 - 2\ v - 3\ w^2 + 2\ w + 3,$$
$$3\ v^2\ z - 1\Big\}\Big]$$

```
Type:  List RegularTriangularSet(Integer, IndexedExponents
OrderedVariableList [b1,x,y,z,t,v,u,w], OrderedVariableList
```

```
[b1,x,y,z,t,v,u,w], NewSparseMultivariatePolynomial(Integer,
                    OrderedVariableList [b1,x,y,z,t,v,u,w]))
```

We retrieve our regular chain of dimension 1 and we get three regular chains of dimension 0 corresponding to the *degenerated cases*. We want now to simplify these zero-dimensional regular chains by using Lazard triangular sets. Moreover, this will allow us to prove that the above decomposition has no redundant component. **N.B.** Generally, decompositions computed by the `REGSET` constructor do not have redundant components. However, to be sure that no redundant component occurs one needs to use the `SREGSET` or `LAZM3PK` constructors.

So let us solve the input system in the sense of Lazard by means of the `LAZM3PK` constructor:

`zeroSetSplit(lf,false)$pack`

$$\Big[\Big\{729\ u^6 + \left(-1458\ w^3 + 729\ w^2 - 4158\ w - 1685\right)\ u^4 +$$
$$\left(729\ w^6 - 1458\ w^5 - 2619\ w^4 - 4892\ w^3 - 297\ w^2 + 5814\ w + 427\right)\ u^2 +$$
$$729\ w^8 + 216\ w^7 - 2900\ w^6 - 2376\ w^5 + 3870\ w^4 + 4072\ w^3 -$$
$$1188\ w^2 - 1656\ w + 529,$$
$$\left(2187\ u^4 + \left(-4374\ w^3 - 972\ w^2 - 12474\ w - 2868\right)\ u^2 +$$
$$2187\ w^6 - 1944\ w^5 - 10125\ w^4 - 4800\ w^3 + 2501\ w^2 + 4968\ w - 1587\right)\ v +$$
$$\left(1944\ w^3 - 108\ w^2\right)\ u^2 + 972\ w^6 + 3024\ w^5 - 1080\ w^4 + 496\ w^3 + 1116\ w^2,$$
$$\left(3\ v^2 + 2\ v - 2\ w\right)\ t - 3\ u\ v^2,$$
$$\left(\left(4\ v - 4\ w\right)\ t - 6\ u\ v^2\right)\ z^2 + \left(2\ t + 3\ v^2\right)\ z - 1\Big\},$$
$$\Big\{81\ w^2 + 18\ w + 28, 729\ u^2 - 1890\ w - 533, 81\ v^2 + \left(-162\ w + 27\right)\ v -$$
$$72\ w - 112,$$
$$11881\ t + \left(972\ w + 2997\right)\ u\ v + \left(-11448\ w - 11536\right)\ u,$$
$$641237934604288\ z^2 + \left(\left(\left(78614584763904\ w + 26785578742272\right)\ u +$$
$$236143618655616\ w + 70221988585728\right)\ v + \left(358520253138432\ w +$$
$$101922133759488\right)\ u + 142598803536000\ w + 54166419595008\right)\ z +$$
$$\left(32655103844499\ w - 44224572465882\right)\ u\ v +$$
$$\left(43213900115457\ w - 32432039102070\right)\ u\Big\},$$
$$\Big\{27\ w^4 + 4\ w^3 - 54\ w^2 - 36\ w + 23, u, 218\ v - 162\ w^3 + 3\ w^2 + 160\ w + 153,$$
$$109\ t^2 - 27\ w^3 - 54\ w^2 + 63\ w + 80,$$
$$1744\ z + \left(-1458\ w^3 + 27\ w^2 + 1440\ w + 505\right)\ t\Big\},$$
$$\Big\{27\ w^4 + 4\ w^3 - 54\ w^2 - 36\ w + 23, u, 218\ v - 162\ w^3 + 3\ w^2 + 160\ w + 153,$$
$$109\ t^2 - 27\ w^3 - 54\ w^2 + 63\ w + 80, 1308\ z + 162\ w^3 - 3\ w^2 - 814\ w - 153\Big\},$$
$$\Big\{729\ w^4 + 972\ w^3 - 1026\ w^2 + 1684\ w + 765, 81\ u^2 + 72\ w^2 + 16\ w - 72,$$
$$702\ v - 162\ w^3 - 225\ w^2 + 40\ w - 99,$$
$$11336\ t + \left(324\ w^3 - 603\ w^2 - 1718\ w - 1557\right)\ u,$$
$$595003968\ z^2 + \left(\left(-963325386\ w^3 - 898607682\ w^2 + 1516286466\ w -$$
$$3239166186\right)\ u - 1579048992\ w^3 - 1796454288\ w^2 + 2428328160\ w -$$
$$4368495024\right)\ z + \left(9713133306\ w^3 + 9678670317\ w^2 - 16726834476\ w +$$
$$28144233593\right)\ u\Big\}\Big]$$

```
         Type:  List SquareFreeRegularTriangularSet(Integer,
   IndexedExponents OrderedVariableList [b1,x,y,z,t,v,u,w],
                    OrderedVariableList [b1,x,y,z,t,v,u,w],
NewSparseMultivariatePolynomial(Integer, OrderedVariableList
                                    [b1,x,y,z,t,v,u,w]))
```

Due to square-free factorization, we obtained now four zero-dimensional regular chains. Moreover, each of them is normalized (the initials are constant). Note that these zero-dimensional components may be investigated further with the `ZeroDimensionalSolvePackage` package constructor.

9.41   Library

The `Library` domain provides a simple way to store Axiom values in a file. This domain is similar to `KeyedAccessFile` but fewer declarations are needed and items of different types can be saved together in the same file.

To create a library, you supply a file name.

```
stuff := library "/tmp/Neat.stuff"
```

$$\text{"/tmp/Neat.stuff"}$$

```
                                                  Type:  Library
```

Now values can be saved by key in the file. The keys should be mnemonic, just as the field names are for records. They can be given either as strings or symbols.

```
stuff.int := 32**2
```

$$1024$$

```
                                          Type:  PositiveInteger
```

```
stuff."poly" := x**2 + 1
```

$$x^2 + 1$$

```
                                          Type:  Polynomial Integer
```

```
stuff.str := "Hello"
```

```
                              "Hello"
```

$$\text{Type: String}$$

You obtain the set of available keys using the **keys** operation.

```
keys stuff
```

$$\left[\texttt{"str"}, \texttt{"poly"}, \texttt{"int"}\right]$$

$$\text{Type: List String}$$

You extract values by giving the desired key in this way.

```
stuff.poly
```

$$x^2 + 1$$

$$\text{Type: Polynomial Integer}$$

```
stuff("poly")
```

$$x^2 + 1$$

$$\text{Type: Polynomial Integer}$$

When the file is no longer needed, you should remove it from the file system.

```
)system rm -rf /tmp/Neat.stuff
```

For more information on related topics, see 9.24 on page 555, 9.81 on page 820, and 9.38 on page 608.

9.42 LieExponentials

```
a: Symbol := 'a
```

$$a$$

$$\text{Type: Symbol}$$

```
b:  Symbol := 'b
```

$$b$$

<div align="right">Type: Symbol</div>

Declarations of domains

```
coef := Fraction(Integer)
```

<div align="center">Fraction Integer</div>

<div align="right">Type: Domain</div>

```
group := LieExponentials(Symbol, coef, 3)
```

<div align="center">LieExponentials(Symbol,Fraction Integer,3)</div>

<div align="right">Type: Domain</div>

```
lpoly := LiePolynomial(Symbol, coef)
```

<div align="center">LiePolynomial(Symbol,Fraction Integer)</div>

<div align="right">Type: Domain</div>

```
poly := XPBWPolynomial(Symbol, coef)
```

<div align="center">XPBWPolynomial(Symbol,Fraction Integer)</div>

<div align="right">Type: Domain</div>

Calculations

```
ea := exp(a::lpoly)$group
```

$$e^{[a]}$$

<div align="right">Type: LieExponentials(Symbol,Fraction Integer,3)</div>

```
eb := exp(b::lpoly)$group
```

$$e^{[b]}$$

Type: LieExponentials(Symbol,Fraction Integer,3)

```
g:  group := ea*eb
```

$$e^{[b]} \; e^{\left(\frac{1}{2} \; [a \; b^2]\right)} \; e^{[a \; b]} \; e^{\left(\frac{1}{2} \; [a^2 \; b]\right)} \; e^{[a]}$$

Type: LieExponentials(Symbol,Fraction Integer,3)

```
g ::  poly
```

$$1 + [a] + [b] + \tfrac{1}{2} \; [a] \; [a] + [a \; b] + [b] \; [a] + \tfrac{1}{2} \; [b] \; [b] + \tfrac{1}{6} \; [a] \; [a] \; [a] + \tfrac{1}{2} \; [a^2 \; b] +$$

$$[a \; b] \; [a] + \frac{1}{2} \; [a \; b^2] + \frac{1}{2} \; [b] \; [a] \; [a] + [b] \; [a \; b] + \frac{1}{2} \; [b] \; [b] \; [a] + \frac{1}{6} \; [b] \; [b] \; [b]$$

Type: XPBWPolynomial(Symbol,Fraction Integer)

```
log(g)$group
```

$$[a] + [b] + \frac{1}{2} \; [a \; b] + \frac{1}{12} \; [a^2 \; b] + \frac{1}{12} \; [a \; b^2]$$

Type: LiePolynomial(Symbol,Fraction Integer)

```
g1:  group := inv(g)
```

$$e^{(-[b])} \; e^{(-[a])}$$

Type: LieExponentials(Symbol,Fraction Integer,3)

```
g*g1
```

$$1$$

Type: LieExponentials(Symbol,Fraction Integer,3)

9.43   LiePolynomial

Declaration of domains

```
RN := Fraction Integer
```

$$\text{Fraction Integer}$$

<div align="right">Type: Domain</div>

```
Lpoly := LiePolynomial(Symbol,RN)
```

$$\text{LiePolynomial(Symbol,Fraction Integer)}$$

<div align="right">Type: Domain</div>

```
Dpoly := XDPOLY(Symbol,RN)
```

$$\text{XDistributedPolynomial(Symbol,Fraction Integer)}$$

<div align="right">Type: Domain</div>

```
Lword := LyndonWord Symbol
```

$$\text{LyndonWord Symbol}$$

<div align="right">Type: Domain</div>

Initialisation

```
a:Symbol := 'a
```

$$a$$

<div align="right">Type: Symbol</div>

```
b:Symbol := 'b
```

$$b$$

$$\text{Type:} \quad \text{Symbol}$$

c:Symbol := 'c

$$c$$

$$\text{Type:} \quad \text{Symbol}$$

aa: Lpoly := a

$$[a]$$

$$\text{Type:} \quad \text{LiePolynomial(Symbol,Fraction Integer)}$$

bb: Lpoly := b

$$[b]$$

$$\text{Type:} \quad \text{LiePolynomial(Symbol,Fraction Integer)}$$

cc: Lpoly := c

$$[c]$$

$$\text{Type:} \quad \text{LiePolynomial(Symbol,Fraction Integer)}$$

p : Lpoly := [aa,bb]

$$[a\ b]$$

$$\text{Type:} \quad \text{LiePolynomial(Symbol,Fraction Integer)}$$

q : Lpoly := [p,bb]

$$\left[a\ b^2\right]$$

$$\text{Type:} \quad \text{LiePolynomial(Symbol,Fraction Integer)}$$

All the Lyndon words of order 4

```
liste :  List Lword := LyndonWordsList([a,b], 4)
```

$$[[a], [b], [a\ b], [a^2\ b], [a\ b^2], [a^3\ b], [a^2\ b^2], [a\ b^3]]$$

<div align="right">Type: List LyndonWord Symbol</div>

```
r:  Lpoly := p + q + 3*LiePoly(liste.4)$Lpoly
```

$$[a\ b] + 3\ [a^2\ b] + [a\ b^2]$$

<div align="right">Type: LiePolynomial(Symbol,Fraction Integer)</div>

```
s:Lpoly := [p,r]
```

$$-3\ [a^2\ b\ a\ b] + [a\ b\ a\ b^2]$$

<div align="right">Type: LiePolynomial(Symbol,Fraction Integer)</div>

```
t:Lpoly := s + 2*LiePoly(liste.3) - 5*LiePoly(liste.5)
```

$$2\ [a\ b] - 5\ [a\ b^2] - 3\ [a^2\ b\ a\ b] + [a\ b\ a\ b^2]$$

<div align="right">Type: LiePolynomial(Symbol,Fraction Integer)</div>

```
degree t
```

$$5$$

<div align="right">Type: PositiveInteger</div>

```
mirror t
```

$$-2\ [a\ b] - 5\ [a\ b^2] - 3\ [a^2\ b\ a\ b] + [a\ b\ a\ b^2]$$

<div align="right">Type: LiePolynomial(Symbol,Fraction Integer)</div>

Jacobi Relation

```
Jacobi(p:  Lpoly, q:  Lpoly, r:  Lpoly): Lpoly == [ [p,q]$Lpoly,
r] + [ [q,r]$Lpoly, p] + [ [r,p]$Lpoly, q]
```

```
Function declaration Jacobi : (
  LiePolynomial(Symbol,  Fraction Integer),
  LiePolynomial(Symbol,Fraction Integer),
  LiePolynomial(Symbol,Fraction Integer)) ->
    LiePolynomial(Symbol,Fraction Integer)
  has been added to workspace.
```

$$\text{Void}$$

Tests

```
test:  Lpoly := Jacobi(a,b,b)
```

$$0$$

$$\text{Type: LiePolynomial(Symbol,Fraction Integer)}$$

```
test:  Lpoly := Jacobi(p,q,r)
```

$$0$$

$$\text{Type: LiePolynomial(Symbol,Fraction Integer)}$$

```
test:  Lpoly := Jacobi(r,s,t)
```

$$0$$

$$\text{Type: LiePolynomial(Symbol,Fraction Integer)}$$

Evaluation

```
eval(p, a, p)$Lpoly
```

$$\left[a\ b^2 \right]$$

$$\text{Type: LiePolynomial(Symbol,Fraction Integer)}$$

```
eval(p, [a,b], [2*bb, 3*aa])$Lpoly
```

$$-6\ [a\ b]$$

<div align="center">

Type: LiePolynomial(Symbol,Fraction Integer)

</div>

```
r:  Lpoly := [p,c]
```

$$[a\ b\ c] + [a\ c\ b]$$

<div align="center">

Type: LiePolynomial(Symbol,Fraction Integer)

</div>

```
r1:  Lpoly := eval(r, [a,b,c], [bb, cc, aa])$Lpoly
```

$$-[a\ b\ c]$$

<div align="center">

Type: LiePolynomial(Symbol,Fraction Integer)

</div>

```
r2:  Lpoly := eval(r, [a,b,c], [cc, aa, bb])$Lpoly
```

$$-[a\ c\ b]$$

<div align="center">

Type: LiePolynomial(Symbol,Fraction Integer)

</div>

```
r + r1 + r2
```

$$0$$

<div align="center">

Type: LiePolynomial(Symbol,Fraction Integer)

</div>

9.44   LinearOrdinaryDifferentialOperator

`LinearOrdinaryDifferentialOperator(A, diff)` is the domain of linear ordinary differential operators with coefficients in a ring `A` with a given derivation.

9.44.1   Differential Operators with Series Coefficients

Problem: Find the first few coefficients of `exp(x)/x**i` of `Dop phi` where

```
Dop := D**3 + G/x**2 * D + H/x**3 - 1
phi := sum(s[i]*exp(x)/x**i, i = 0..)
```

Solution:

Define the differential.

```
Dx:  LODO(EXPR INT, f +-> D(f, x))
```

<div align="right">Void</div>

```
Dx := D()
```

$$D$$

<div align="center">Type: LinearOrdinaryDifferentialOperator(Expression
Integer,theMap NIL)</div>

Now define the differential operator Dop.

```
Dop:= Dx**3 + G/x**2*Dx + H/x**3 - 1
```

$$D^3 + \frac{G}{x^2}\, D + \frac{-x^3 + H}{x^3}$$

<div align="center">Type: LinearOrdinaryDifferentialOperator(Expression
Integer,theMap NIL)</div>

```
n == 3
```

<div align="right">Void</div>

```
phi == reduce(+,[subscript(s,[i])*exp(x)/x**i for i in 0..n])
```

<div align="right">Void</div>

```
phi1 == Dop(phi) / exp x
```

<div align="right">Void</div>

```
phi2 == phi1 *x**(n+3)
```

```
                                                                 Void
```

```
phi3 == retract(phi2)@(POLY INT)
```

```
                                                                 Void
```

```
pans == phi3 ::UP(x,POLY INT)
```

```
                                                                 Void
```

```
pans1 == [coefficient(pans, (n+3-i) ::  NNI) for i in 2..n+1]
```

```
                                                                 Void
```

```
leq == solve(pans1,[subscript(s,[i]) for i in 1..n])
```

```
                                                                 Void
```

Evaluate this for several values of n.

```
leq
```

```
   Compiling body of rule n to compute value of type PositiveInteger
   Compiling body of rule phi to compute value of type Expression
      Integer
   Compiling body of rule phi1 to compute value of type Expression
      Integer
   Compiling body of rule phi2 to compute value of type Expression
      Integer
   Compiling body of rule phi3 to compute value of type Polynomial
      Integer
   Compiling body of rule pans to compute value of type
      UnivariatePolynomial(x,Polynomial Integer)
   Compiling body of rule pans1 to compute value of type List
      Polynomial Integer
   Compiling body of rule leq to compute value of type List List
      Equation Fraction Polynomial Integer
   Compiling function G83347 with type Integer -> Boolean
```

```
n==4
```

$$\left[\left[s_1 = \frac{s_0\ G}{3}, s_2 = \frac{3\ s_0\ H + s_0\ G^2 + 6\ s_0\ G}{18},\right.\right.$$

$$\left.\left. s_3 = \frac{(9\ s_0\ G + 54\ s_0)\ H + s_0\ G^3 + 18\ s_0\ G^2 + 72\ s_0\ G}{162}\right]\right]$$

<div align="center">Type: List List Equation Fraction Polynomial Integer</div>

```
leq
```

$$\left[\left[s_1 = \frac{s_0\ G}{3}, s_2 = \frac{3\ s_0\ H + s_0\ G^2 + 6\ s_0\ G}{18},\right.\right.$$

$$\left.\left. s_3 = \frac{(9\ s_0\ G + 54\ s_0)\ H + s_0\ G^3 + 18\ s_0\ G^2 + 72\ s_0\ G}{162}\right]\right]$$

<div align="center">Type: List List Equation Fraction Polynomial Integer</div>

```
n==7
```

```
   Compiled code for n has been cleared.
   Compiled code for leq has been cleared.
   Compiled code for pans1 has been cleared.
   Compiled code for phi2 has been cleared.
   Compiled code for phi has been cleared.
   Compiled code for phi3 has been cleared.
   Compiled code for phi1 has been cleared.
   Compiled code for pans has been cleared.
   1 old definition(s) deleted for function or rule n
```

<div align="right">Void</div>

```
leq
```

```
Compiling body of rule n to compute value of type PositiveInteger
```

```
+++ |*0;n;1;G82322| redefined
Compiling body of rule phi to compute value of type Expression
  Integer
```

```
+++ |*0;phi;1;G82322| redefined
Compiling body of rule phi1 to compute value of type Expression
  Integer

+++ |*0;phi1;1;G82322| redefined
Compiling body of rule phi2 to compute value of type Expression
  Integer

+++ |*0;phi2;1;G82322| redefined
Compiling body of rule phi3 to compute value of type Polynomial
  Integer

+++ |*0;phi3;1;G82322| redefined
Compiling body of rule pans to compute value of type
  UnivariatePolynomial(x,Polynomial Integer)

+++ |*0;pans;1;G82322| redefined
Compiling body of rule pans1 to compute value of type List
  Polynomial Integer

+++ |*0;pans1;1;G82322| redefined
Compiling body of rule leq to compute value of type List List
  Equation Fraction Polynomial Integer

+++ |*0;leq;1;G82322| redefined
```

$$\left[\left[s_1 = \frac{s_0 \ G}{3},\right.\right.$$

$$s_2 = \frac{3 \ s_0 \ H + s_0 \ G^2 + 6 \ s_0 \ G}{18},$$

$$s_3 = \frac{(9 \ s_0 \ G + 54 \ s_0) \ H + s_0 \ G^3 + 18 \ s_0 \ G^2 + 72 \ s_0 \ G}{162},$$

$$s_4 = \frac{\begin{pmatrix} 27 \ s_0 \ H^2 + \left(18 \ s_0 \ G^2 + 378 \ s_0 \ G + 1296 \ s_0\right) \ H + \\ s_0 \ G^4 + 36 \ s_0 \ G^3 + 396 \ s_0 \ G^2 + 1296 \ s_0 \ G \end{pmatrix}}{1944},$$

$$s_5 = \frac{\begin{pmatrix} \left(135 \ s_0 \ G + 2268 \ s_0\right) \ H^2 + \\ \left(30 \ s_0 \ G^3 + 1350 \ s_0 \ G^2 + 16416 \ s_0 \ G + 38880 \ s_0\right) \ H + \\ s_0 \ G^5 + 60 \ s_0 \ G^4 + 1188 \ s_0 \ G^3 + 9504 \ s_0 \ G^2 + 25920 \ s_0 \ G \end{pmatrix}}{29160},$$

$$s_6 = \frac{\begin{pmatrix} 405 \ s_0 \ H^3 + \\[6pt] \left(405 \ s_0 \ G^2 + 18468 \ s_0 \ G + 174960 \ s_0\right) \ H^2 + \\[6pt] \left(45 \ s_0 \ G^4 + 3510 \ s_0 \ G^3 + 88776 \ s_0 \ G^2 + 777600 \ s_0 \ G + \right. \\ 1166400 \ s_0) \ H + \\[6pt] s_0 \ G^6 + 90 \ s_0 \ G^5 + 2628 \ s_0 \ G^4 + 27864 \ s_0 \ G^3 + 90720 \ s_0 \ G^2 \end{pmatrix}}{524880},$$

$$s_7 = \frac{\begin{pmatrix} \left(2835 \ s_0 \ G + 91854 \ s_0\right) \ H^3 + \\[6pt] \left(945 \ s_0 \ G^3 + 81648 \ s_0 \ G^2 + 2082996 \ s_0 \ G + 14171760 \ s_0\right) \ H^2 + \\[6pt] \left(63 \ s_0 \ G^5 + 7560 \ s_0 \ G^4 + 317520 \ s_0 \ G^3 + 5554008 \ s_0 \ G^2 + \right. \\ 34058880 \ s_0 \ G) \ H + \\[6pt] s_0 \ G^7 + 126 \ s_0 \ G^6 + 4788 \ s_0 \ G^5 + 25272 \ s_0 \ G^4 - 1744416 \ s_0 \ G^3 - \\ 26827200 \ s_0 \ G^2 - 97977600 \ s_0 \ G \end{pmatrix}}{11022480}$$

```
Type:   List List Equation Fraction Polynomial Integer
```

9.45   LinearOrdinaryDifferentialOperator1

`LinearOrdinaryDifferentialOperator1(A)` is the domain of linear ordinary differential operators with coefficients in the differential ring `A`.

9.45.1   Differential Operators with Rational Function Coefficients

This example shows differential operators with rational function coefficients. In this case operator multiplication is non-commutative and, since the coefficients form a field, an operator division algorithm exists.

We begin by defining `RFZ` to be the rational functions in `x` with integer coefficients and `Dx` to be the differential operator for `d/dx`.

```
RFZ := Fraction UnivariatePolynomial('x, Integer)
```

$$\text{Fraction UnivariatePolynomial(x,Integer)}$$

<div align="right">Type: Domain</div>

```
x :  RFZ := 'x
```

$$x$$

<div align="right">Type: Fraction UnivariatePolynomial(x,Integer)</div>

```
Dx :  LODO1 RFZ := D()
```

$$D$$

<div align="right">Type: LinearOrdinaryDifferentialOperator1 Fraction
UnivariatePolynomial(x,Integer)</div>

Operators are created using the usual arithmetic operations.

```
b :  LODO1 RFZ := 3*x**2*Dx**2 + 2*Dx + 1/x
```

$$3 \ x^2 \ D^2 + 2 \ D + \frac{1}{x}$$

<div align="right">Type: LinearOrdinaryDifferentialOperator1 Fraction
UnivariatePolynomial(x,Integer)</div>

```
a :  LODO1 RFZ := b*(5*x*Dx + 7)
```

$$15 \ x^3 \ D^3 + \left(51 \ x^2 + 10 \ x\right) \ D^2 + 29 \ D + \frac{7}{x}$$

<div align="right">Type: LinearOrdinaryDifferentialOperator1 Fraction
UnivariatePolynomial(x,Integer)</div>

Operator multiplication corresponds to functional composition.

```
p := x**2 + 1/x**2
```

$$\frac{x^4 + 1}{x^2}$$

Since operator coefficients depend on x, the multiplication is not commutative.

```
(a*b - b*a) p
```

$$\frac{-75\ x^4 + 540\ x - 75}{x^4}$$

When the coefficients of operator polynomials come from a field, as in this case, it is possible to define operator division. Division on the left and division on the right yield different results when the multiplication is non-commutative.

The results of **leftDivide** and **rightDivide** are quotient-remainder pairs satisfying:

```
leftDivide(a,b) = [q, r] such that a = b*q + r
rightDivide(a,b) = [q, r] such that a = q*b + r
```

In both cases, the **degree** of the remainder, r, is less than the degree of b.

```
ld := leftDivide(a,b)
```

$$[quotient = 5\ x\ D + 7, remainder = 0]$$

```
a = b * ld.quotient + ld.remainder
```

$$15\ x^3\ D^3 + \left(51\ x^2 + 10\ x\right)\ D^2 + 29\ D + \tfrac{7}{x} =$$

$$15\ x^3\ D^3 + \left(51\ x^2 + 10\ x\right)\ D^2 + 29\ D + \frac{7}{x}$$

The operations of left and right division are so-called because the quotient is obtained by dividing a on that side by b.

```
rd := rightDivide(a,b)
```

$$\left[quotient = 5 \ x \ D + 7, remainder = 10 \ D + \frac{5}{x}\right]$$

```
        Type:   Record(quotient:  LinearOrdinaryDifferentialOperator1
                 Fraction UnivariatePolynomial(x,Integer), remainder:
                         LinearOrdinaryDifferentialOperator1 Fraction
                                        UnivariatePolynomial(x,Integer))
```

```
a = rd.quotient * b + rd.remainder
```

$$15 \ x^3 \ D^3 + \left(51 \ x^2 + 10 \ x\right) D^2 + 29 \ D + \frac{7}{x} =$$

$$15 \ x^3 \ D^3 + \left(51 \ x^2 + 10 \ x\right) D^2 + 29 \ D + \frac{7}{x}$$

```
        Type:   Equation LinearOrdinaryDifferentialOperator1 Fraction
                                        UnivariatePolynomial(x,Integer)
```

Operations **rightQuotient** and **rightRemainder** are available if only one of the quotient or remainder are of interest to you. This is the quotient from right division.

```
rightQuotient(a,b)
```

$$5 \ x \ D + 7$$

```
        Type:   LinearOrdinaryDifferentialOperator1 Fraction
                                        UnivariatePolynomial(x,Integer)
```

This is the remainder from right division. The corresponding "left" functions **leftQuotient** and **leftRemainder** are also available.

```
rightRemainder(a,b)
```

$$10 \ D + \frac{5}{x}$$

```
        Type:   LinearOrdinaryDifferentialOperator1 Fraction
                                        UnivariatePolynomial(x,Integer)
```

For exact division, the operations **leftExactQuotient** and **rightExactQuotient** are supplied. These return the quotient but only if the remainder is zero. The call `rightExactQuotient(a,b)` would yield an error.

leftExactQuotient(a,b)

$$5 \ x \ D + 7$$

Type: Union(LinearOrdinaryDifferentialOperator1 Fraction
UnivariatePolynomial(x,Integer),...)

The division operations allow the computation of left and right greatest common divisors (**leftGcd** and **rightGcd**) via remainder sequences, and consequently the computation of left and right least common multiples (**rightLcm** and **left-Lcm**).

e := leftGcd(a,b)

$$3 \ x^2 \ D^2 + 2 \ D + \frac{1}{x}$$

Type: LinearOrdinaryDifferentialOperator1 Fraction
UnivariatePolynomial(x,Integer)

Note that a greatest common divisor doesn't necessarily divide a and b on both sides. Here the left greatest common divisor does not divide a on the right.

leftRemainder(a, e)

$$0$$

Type: LinearOrdinaryDifferentialOperator1 Fraction
UnivariatePolynomial(x,Integer)

rightRemainder(a, e)

$$10 \ D + \frac{5}{x}$$

Type: LinearOrdinaryDifferentialOperator1 Fraction
UnivariatePolynomial(x,Integer)

Similarly, a least common multiple is not necessarily divisible from both sides.

f := rightLcm(a,b)

$$15 \ x^3 \ D^3 + \left(51 \ x^2 + 10 \ x\right) \ D^2 + 29 \ D + \frac{7}{x}$$

<div align="right">
Type: LinearOrdinaryDifferentialOperator1 Fraction
UnivariatePolynomial(x,Integer)
</div>

`rightRemainder(f, b)`

$$10\ D + \frac{5}{x}$$

<div align="right">
Type: LinearOrdinaryDifferentialOperator1 Fraction
UnivariatePolynomial(x,Integer)
</div>

`leftRemainder(f, b)`

$$0$$

<div align="right">
Type: LinearOrdinaryDifferentialOperator1 Fraction
UnivariatePolynomial(x,Integer)
</div>

9.46   LinearOrdinaryDifferentialOperator2

`LinearOrdinaryDifferentialOperator2(A, M)` is the domain of linear ordinary differential operators with coefficients in the differential ring `A` and operating on `M`, an `A`-module. This includes the cases of operators which are polynomials in `D` acting upon scalar or vector expressions of a single variable. The coefficients of the operator polynomials can be integers, rational functions, matrices or elements of other domains.

9.46.1   Differential Operators with Constant Coefficients

This example shows differential operators with rational number coefficients operating on univariate polynomials.

We begin by making type assignments so we can conveniently refer to univariate polynomials in `x` over the rationals.

`Q := Fraction Integer`

<div align="center">
Fraction Integer
</div>

<div align="right">
Type: Domain
</div>

```
PQ := UnivariatePolynomial('x, Q)
```

$$\text{UnivariatePolynomial(x,Fraction Integer)}$$

<div align="right">Type: Domain</div>

```
x:  PQ := 'x
```

$$x$$

<div align="right">Type: UnivariatePolynomial(x,Fraction Integer)</div>

Now we assign `Dx` to be the differential operator **D** corresponding to `d/dx`.

```
Dx:  LODO2(Q, PQ) := D()
```

$$D$$

<div align="right">Type: LinearOrdinaryDifferentialOperator2(Fraction Integer,
UnivariatePolynomial(x,Fraction Integer))</div>

New operators are created as polynomials in `D()`.

```
a := Dx + 1
```

$$D + 1$$

<div align="right">Type: LinearOrdinaryDifferentialOperator2(Fraction Integer,
UnivariatePolynomial(x,Fraction Integer))</div>

```
b := a + 1/2*Dx**2 - 1/2
```

$$\frac{1}{2} \, D^2 + D + \frac{1}{2}$$

<div align="right">Type: LinearOrdinaryDifferentialOperator2(Fraction Integer,
UnivariatePolynomial(x,Fraction Integer))</div>

To apply the operator `a` to the value `p` the usual function call syntax is used.

```
p := 4*x**2 + 2/3
```

$$4\,x^2 + \frac{2}{3}$$

<div align="center">Type: UnivariatePolynomial(x,Fraction Integer)</div>

a p

$$4\,x^2 + 8\,x + \frac{2}{3}$$

<div align="center">Type: UnivariatePolynomial(x,Fraction Integer)</div>

Operator multiplication is defined by the identity `(a*b) p = a(b(p))`

(a * b) p = a b p

$$2\,x^2 + 12\,x + \frac{37}{3} = 2\,x^2 + 12\,x + \frac{37}{3}$$

<div align="center">Type: Equation UnivariatePolynomial(x,Fraction Integer)</div>

Exponentiation follows from multiplication.

c := (1/9)*b*(a + b)**2

$$\frac{1}{72}\,D^6 + \frac{5}{36}\,D^5 + \frac{13}{24}\,D^4 + \frac{19}{18}\,D^3 + \frac{79}{72}\,D^2 + \frac{7}{12}\,D + \frac{1}{8}$$

<div align="center">Type: LinearOrdinaryDifferentialOperator2(Fraction Integer,
UnivariatePolynomial(x,Fraction Integer))</div>

Finally, note that operator expressions may be applied directly.

(a**2 - 3/4*b + c) (p + 1)

$$3\,x^2 + \frac{44}{3}\,x + \frac{541}{36}$$

<div align="center">Type: UnivariatePolynomial(x,Fraction Integer)</div>

9.46.2 Differential Operators with Matrix Coefficients Operating on Vectors

This is another example of linear ordinary differential operators with non-commutative multiplication. Unlike the rational function case, the differential ring of square matrices (of a given dimension) with univariate polynomial entries does not form a field. Thus the number of operations available is more limited.

In this section, the operators have three by three matrix coefficients with polynomial entries.

```
PZ := UnivariatePolynomial(x,Integer)
```

$$UnivariatePolynomial(x, Integer)$$

                                                    Type: Domain

```
x:PZ := 'x
```

$$x$$

                            Type: UnivariatePolynomial(x,Integer)

```
Mat := SquareMatrix(3,PZ)
```

$$SquareMatrix(3, UnivariatePolynomial(x, Integer))$$

                                                    Type: Domain

The operators act on the vectors considered as a `Mat`-module.

```
Vect := DPMM(3, PZ, Mat, PZ)
```

$$DirectProductMatrixModule(3,$$
$$UnivariatePolynomial(x, Integer),$$
$$SquareMatrix(3, UnivariatePolynomial(x, Integer)),$$
$$UnivariatePolynomial(x, Integer))$$

                                                    Type: Domain

```
Modo := LODO2(Mat, Vect)
```

LinearOrdinaryDifferentialOperator2(
SquareMatrix(3, UnivariatePolynomial(x, Integer)),
DirectProductMatrixModule(3,
UnivariatePolynomial(x, Integer),
SquareMatrix(3, UnivariatePolynomial(x, Integer)),
UnivariatePolynomial(x, Integer)))

                                                    Type: Domain

The matrix `m` is used as a coefficient and the vectors `p` and `q` are operated upon.

```
m:Mat := matrix [ [x**2,1,0],[1,x**4,0],[0,0,4*x**2] ]
```

$$
\begin{bmatrix}
x^2 & 1 & 0 \\
1 & x^4 & 0 \\
0 & 0 & 4 \; x^2
\end{bmatrix}
$$

          Type: SquareMatrix(3,UnivariatePolynomial(x,Integer))

```
p:Vect := directProduct [3*x**2+1,2*x,7*x**3+2*x]
```

$$
\left[3 \; x^2 + 1, 2 \; x, 7 \; x^3 + 2 \; x \right]
$$

                          Type: DirectProductMatrixModule(3,
                                    UnivariatePolynomial(x,Integer),
                  SquareMatrix(3,UnivariatePolynomial(x,Integer)),
                                    UnivariatePolynomial(x,Integer))

```
q:  Vect := m * p
```

$$
\left[3 \; x^4 + x^2 + 2 \; x, 2 \; x^5 + 3 \; x^2 + 1, 28 \; x^5 + 8 \; x^3 \right]
$$

                          Type: DirectProductMatrixModule(3,
                                    UnivariatePolynomial(x,Integer),
                  SquareMatrix(3,UnivariatePolynomial(x,Integer)),
                                    UnivariatePolynomial(x,Integer))

Now form a few operators.

```
Dx :  Modo := D()
```

$$
D
$$

Type: LinearOrdinaryDifferentialOperator2(
SquareMatrix(3,UnivariatePolynomial(x,Integer)),
DirectProductMatrixModule(3, UnivariatePolynomial(x,Integer),
SquareMatrix(3,UnivariatePolynomial(x,Integer)),
UnivariatePolynomial(x,Integer)))

a : Modo := Dx + m

$$D + \begin{bmatrix} x^2 & 1 & 0 \\ 1 & x^4 & 0 \\ 0 & 0 & 4\,x^2 \end{bmatrix}$$

Type: LinearOrdinaryDifferentialOperator2(
SquareMatrix(3,UnivariatePolynomial(x,Integer)),
DirectProductMatrixModule(3, UnivariatePolynomial(x,Integer),
SquareMatrix(3, UnivariatePolynomial(x,Integer)),
UnivariatePolynomial(x,Integer)))

b : Modo := m*Dx + 1

$$\begin{bmatrix} x^2 & 1 & 0 \\ 1 & x^4 & 0 \\ 0 & 0 & 4\,x^2 \end{bmatrix} D + \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Type: LinearOrdinaryDifferentialOperator2(SquareMatrix(3,
UnivariatePolynomial(x,Integer)), DirectProductMatrixModule(3,
UnivariatePolynomial(x,Integer), SquareMatrix(3,
UnivariatePolynomial(x,Integer)),
UnivariatePolynomial(x,Integer)))

c := a*b

$$\begin{bmatrix} x^2 & 1 & 0 \\ 1 & x^4 & 0 \\ 0 & 0 & 4\,x^2 \end{bmatrix} D^2 +$$

$$\begin{bmatrix} x^4 + 2\,x + 2 & x^4 + x^2 & 0 \\ x^4 + x^2 & x^8 + 4\,x^3 + 2 & 0 \\ 0 & 0 & 16\,x^4 + 8\,x + 1 \end{bmatrix} D +$$

$$\begin{bmatrix} x^2 & 1 & 0 \\ 1 & x^4 & 0 \\ 0 & 0 & 4\,x^2 \end{bmatrix}$$

```
      Type:  LinearOrdinaryDifferentialOperator2( SquareMatrix(3,
  UnivariatePolynomial(x,Integer)), DirectProductMatrixModule(3,
              UnivariatePolynomial(x,Integer), SquareMatrix(3,
                          UnivariatePolynomial(x,Integer)),
                          UnivariatePolynomial(x,Integer)))
```

These operators can be applied to vector values.

`a p`

$$\left[3\ x^4 + x^2 + 8\ x, 2\ x^5 + 3\ x^2 + 3, 28\ x^5 + 8\ x^3 + 21\ x^2 + 2\right]$$

```
                      Type:  DirectProductMatrixModule(3,
          UnivariatePolynomial(x,Integer), SquareMatrix(3,
                      UnivariatePolynomial(x,Integer)),
                      UnivariatePolynomial(x,Integer))
```

`b p`

$$\left[6\ x^3 + 3\ x^2 + 3, 2\ x^4 + 8\ x, 84\ x^4 + 7\ x^3 + 8\ x^2 + 2\ x\right]$$

```
                      Type:  DirectProductMatrixModule(3,
          UnivariatePolynomial(x,Integer), SquareMatrix(3,
                      UnivariatePolynomial(x,Integer)),
                      UnivariatePolynomial(x,Integer))
```

`(a + b + c) (p + q)`

$$\left[10\ x^8 + 12\ x^7 + 16\ x^6 + 30\ x^5 + 85\ x^4 + 94\ x^3 + 40\ x^2 + 40\ x + 17,\right.$$

$$10\ x^{12} + 10\ x^9 + 12\ x^8 + 92\ x^7 + 6\ x^6 + 32\ x^5 + 72\ x^4 + 28\ x^3 + 49\ x^2 + 32\ x + 19,$$

$$\left.2240\ x^8 + 224\ x^7 + 1280\ x^6 + 3508\ x^5 + 492\ x^4 + 751\ x^3 + 98\ x^2 + 18\ x + 4\right]$$

```
                      Type:  DirectProductMatrixModule(3,
          UnivariatePolynomial(x,Integer), SquareMatrix(3,
                      UnivariatePolynomial(x,Integer)),
                      UnivariatePolynomial(x,Integer))
```

9.47 List

A is a finite collection of elements in a specified order that can contain duplicates. A list is a convenient structure to work with because it is easy to add or remove elements and the length need not be constant. There are many different kinds of lists in Axiom, but the default types (and those used most often) are created by the `List` constructor. For example, there are objects of type `List Integer`, `List Float` and `List Polynomial Fraction Integer`. Indeed, you can even have `List List List Boolean` (that is, lists of lists of lists of Boolean values). You can have lists of any type of Axiom object.

9.47.1 Creating Lists

The easiest way to create a list with, for example, the elements `2, 4, 5, 6` is to enclose the elements with square brackets and separate the elements with commas.

The spaces after the commas are optional, but they do improve the readability.

```
[2, 4, 5, 6]
```

$$[2, 4, 5, 6]$$

                                Type: List PositiveInteger

To create a list with the single element `1`, you can use either `[1]` or the operation **list**.

```
[1]
```

$$[1]$$

                                Type: List PositiveInteger

```
list(1)
```

$$[1]$$

                                Type: List PositiveInteger

Once created, two lists `k` and `m` can be concatenated by issuing `append(k,m)`. **append** does *not* physically join the lists, but rather produces a new list with the elements coming from the two arguments.

```
append([1,2,3],[5,6,7])
```

$$[1, 2, 3, 5, 6, 7]$$

                                        Type: List PositiveInteger

Use **cons** to append an element onto the front of a list.

```
cons(10,[9,8,7])
```

$$[10, 9, 8, 7]$$

                                        Type: List PositiveInteger

9.47.2   Accessing List Elements

To determine whether a list has any elements, use the operation **empty?**.

```
empty?   [x+1]
```

                            false

                                            Type: Boolean

Alternatively, equality with the list constant **nil** can be tested.

```
([] = nil)@Boolean
```

                            true

                                            Type: Boolean

We'll use this in some of the following examples.

```
k := [4,3,7,3,8,5,9,2]
```

$$[4, 3, 7, 3, 8, 5, 9, 2]$$

                                        Type: List PositiveInteger

Each of the next four expressions extracts the **first** element of k.

```
first k
```

$$4$$

                                            Type: PositiveInteger

```
k.first
```

$$4$$

                                            Type: PositiveInteger

```
k.1
```

$$4$$

                                            Type: PositiveInteger

```
k(1)
```

$$4$$

                                            Type: PositiveInteger

The last two forms generalize to `k.i` and `k(i)`, respectively, where $1 \leq i \leq n$ and `n` equals the length of `k`.

This length is calculated by "`#`".

```
n := #k
```

$$8$$

                                            Type: PositiveInteger

Performing an operation such as `k.i` is sometimes referred to as *indexing into k* or *elting into k*. The latter phrase comes about because the name of the operation that extracts elements is called **elt**. That is, `k.3` is just alternative syntax for `elt(k,3)`. It is important to remember that list indices begin with 1. If we issue `k := [1,3,2,9,5]` then `k.4` returns 9. It is an error to use an index that is not in the range from `1` to the length of the list.

The last element of a list is extracted by any of the following three expressions.

`last k`

$$2$$

<div align="right">Type: PositiveInteger</div>

`k.last`

$$2$$

<div align="right">Type: PositiveInteger</div>

This form computes the index of the last element and then extracts the element from the list.

`k.(#k)`

$$2$$

<div align="right">Type: PositiveInteger</div>

9.47.3   Changing List Elements

We'll use this in some of the following examples.

`k := [4,3,7,3,8,5,9,2]`

$$[4, 3, 7, 3, 8, 5, 9, 2]$$

<div align="right">Type: List PositiveInteger</div>

List elements are reset by using the `k.i` form on the left-hand side of an assignment. This expression resets the first element of `k` to `999`.

`k.1 := 999`

$$999$$

<div align="right">Type: PositiveInteger</div>

As with indexing into a list, it is an error to use an index that is not within the proper bounds. Here you see that `k` was modified.

k

$$[999, 3, 7, 3, 8, 5, 9, 2]$$

<div align="right">Type: List PositiveInteger</div>

The operation that performs the assignment of an element to a particular position in a list is called **setelt**. This operation is *destructive* in that it changes the list. In the above example, the assignment returned the value 999 and k was modified. For this reason, lists are called objects: it is possible to change part of a list (mutate it) rather than always returning a new list reflecting the intended modifications.

Moreover, since lists can share structure, changes to one list can sometimes affect others.

k := [1,2]

$$[1, 2]$$

<div align="right">Type: List PositiveInteger</div>

m := cons(0,k)

$$[0, 1, 2]$$

<div align="right">Type: List Integer</div>

Change the second element of m.

m.2 := 99

$$99$$

<div align="right">Type: PositiveInteger</div>

See, m was altered.

m

$$[0, 99, 2]$$

<div align="right">Type: List Integer</div>

But what about k? It changed too!

```
k
```

$$[99, 2]$$

                                        Type: List PositiveInteger

9.47.4   Other Functions

An operation that is used frequently in list processing is that which returns all elements in a list after the first element.

```
k := [1,2,3]
```

$$[1, 2, 3]$$

                                        Type: List PositiveInteger

Use the **rest** operation to do this.

```
rest k
```

$$[2, 3]$$

                                        Type: List PositiveInteger

To remove duplicate elements in a list k, use **removeDuplicates**.

```
removeDuplicates [4,3,4,3,5,3,4]
```

$$[4, 3, 5]$$

                                        Type: List PositiveInteger

To get a list with elements in the order opposite to those in a list k, use **reverse**.

```
reverse [1,2,3,4,5,6]
```

$$[6, 5, 4, 3, 2, 1]$$

To test whether an element is in a list, use **member?**: `member?(a,k)` returns
`true` or `false` depending on whether `a` is in `k` or not.

```
member?(1/2,[3/4,5/6,1/2])
```

true

Type: Boolean

```
member?(1/12,[3/4,5/6,1/2])
```

false

Type: Boolean

As an exercise, the reader should determine how to get a list containing all but
the last of the elements in a given non-empty list `k`.[4]

9.47.5 Dot, Dot

Certain lists are used so often that Axiom provides an easy way of constructing
them. If `n` and `m` are integers, then `expand [n..m]` creates a list containing `n`,
`n+1, ... m`. If `n > m` then the list is empty. It is actually permissible to leave
off the `m` in the dot-dot construction (see below).

The dot-dot notation can be used more than once in a list construction and
with specific elements being given. Items separated by dots are called *segments.*

```
[1..3,10,20..23]
```

$$[1..3, 10..10, 20..23]$$

Type: List Segment PositiveInteger

Segments can be expanded into the range of items between the endpoints by
using **expand**.

```
expand [1..3,10,20..23]
```

---

[4]`reverse(rest(reverse(k)))` works.

$$[1, 2, 3, 10, 20, 21, 22, 23]$$

<div align="right">Type: List Integer</div>

What happens if we leave off a number on the right-hand side of "`..`"?

`expand [1..]`

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \ldots]$$

<div align="right">Type: Stream Integer</div>

What is created in this case is a `Stream` which is a generalization of a list. See 9.76 on page 801 for more information.

9.48   LyndonWord

Initialisations

`a:Symbol :='a`

$$a$$

<div align="right">Type: Symbol</div>

`b:Symbol :='b`

$$b$$

<div align="right">Type: Symbol</div>

`c:Symbol :='c`

$$c$$

<div align="right">Type: Symbol</div>

`lword:= LyndonWord(Symbol)`

<div align="center">LyndonWord Symbol</div>

$$\text{Type: Domain}$$

```
magma := Magma(Symbol)
```

$$\text{Magma Symbol}$$

$$\text{Type: Domain}$$

```
word := OrderedFreeMonoid(Symbol)
```

$$\text{OrderedFreeMonoid Symbol}$$

$$\text{Type: Domain}$$

All Lyndon words of with a, b, c to order 3

```
LyndonWordsList1([a,b,c],3)$lword
```

$$[[[a],[b],[c]],[[a\ b],[a\ c],[b\ c]],$$

$$\left[\left[a^2\ b\right],\left[a^2\ c\right],\left[a\ b^2\right],[a\ b\ c],[a\ c\ b],\left[a\ c^2\right],\left[b^2\ c\right],\left[b\ c^2\right]\right]]$$

$$\text{Type: OneDimensionalArray List LyndonWord Symbol}$$

All Lyndon words of with a, b, c to order 3 in flat list

```
LyndonWordsList([a,b,c],3)$lword
```

$$\left[[a],[b],[c],[a\ b],[a\ c],[b\ c],\left[a^2\ b\right],\left[a^2\ c\right],\left[a\ b^2\right],\right.$$

$$\left.[a\ b\ c],[a\ c\ b],\left[a\ c^2\right],\left[b^2\ c\right],\left[b\ c^2\right]\right]$$

$$\text{Type: List LyndonWord Symbol}$$

All Lyndon words of with a, b to order 5

```
lw := LyndonWordsList([a,b],5)$lword
```

$$\left[[a],[b],[a\ b],\left[a^2\ b\right],\left[a\ b^2\right],\left[a^3\ b\right],\left[a^2\ b^2\right],\left[a\ b^3\right],\left[a^4\ b\right],\right.$$

$$\left.\left[a^3\ b^2\right],\left[a^2\ b\ a\ b\right],\left[a^2\ b^3\right],\left[a\ b\ a\ b^2\right],\left[a\ b^4\right]\right]$$

<div align="right">Type: List LyndonWord Symbol</div>

w1 : word := lw.4 :: word

$$a^2\ b$$

<div align="right">Type: OrderedFreeMonoid Symbol</div>

w2 : word := lw.5 :: word

$$a\ b^2$$

<div align="right">Type: OrderedFreeMonoid Symbol</div>

Let's try factoring

```
factor(a::word)$lword
```

$$[[a]]$$

<div align="right">Type: List LyndonWord Symbol</div>

```
factor(w1*w2)$lword
```

$$\left[\left[a^2\ b\ a\ b^2\right]\right]$$

<div align="right">Type: List LyndonWord Symbol</div>

```
factor(w2*w2)$lword
```

$$\left[\left[a\ b^2\right],\left[a\ b^2\right]\right]$$

<div align="right">Type: List LyndonWord Symbol</div>

```
factor(w2*w1)$lword
```

$$\left[\left[a\ b^2\right],\left[a^2\ b\right]\right]$$

<div align="right">Type: List LyndonWord Symbol</div>

Checks and coercions

```
lyndon?(w1)$lword
```

$$\text{true}$$

Type: Boolean

```
lyndon?(w1*w2)$lword
```

$$\text{true}$$

Type: Boolean

```
lyndon?(w2*w1)$lword
```

$$\text{false}$$

Type: Boolean

```
lyndonIfCan(w1)$lword
```

$$\left[a^2\ b\right]$$

Type: Union(LyndonWord Symbol,...)

```
lyndonIfCan(w2*w1)$lword
```

$$\text{"failed"}$$

Type: Union("failed",...)

```
lyndon(w1)$lword
```

$$\left[a^2\ b\right]$$

Type: LyndonWord Symbol

```
lyndon(w1*w2)$lword
```

$$\left[a^2\ b\ a\ b^2\right]$$

Type: LyndonWord Symbol

9.49   Magma

Initialisations

```
x:Symbol :='x
```

$$x$$

<div align="right">Type: Symbol</div>

```
y:Symbol :='y
```

$$y$$

<div align="right">Type: Symbol</div>

```
z:Symbol :='z
```

$$z$$

<div align="right">Type: Symbol</div>

```
word := OrderedFreeMonoid(Symbol)
```

$$\text{OrderedFreeMonoid Symbol}$$

<div align="right">Type: Domain</div>

```
tree := Magma(Symbol)
```

$$\text{Magma Symbol}$$

<div align="right">Type: Domain</div>

Let's make some trees

```
a:tree := x*x
```

$$[x, x]$$

Type: Magma Symbol

```
b:tree := y*y
```

$$[y, y]$$

Type: Magma Symbol

```
c:tree := a*b
```

$$[[x, x], [y, y]]$$

Type: Magma Symbol

Query the trees

```
left c
```

$$[x, x]$$

Type: Magma Symbol

```
right c
```

$$[y, y]$$

Type: Magma Symbol

```
length c
```

$$4$$

Type: PositiveInteger

Coerce to the monoid

```
c::word
```

$$x^2 \ y^2$$

$$\text{Type: OrderedFreeMonoid Symbol}$$

Check ordering

```
a < b
```

$$\text{true}$$

$$\text{Type: Boolean}$$

```
a < c
```

$$\text{true}$$

$$\text{Type: Boolean}$$

```
b < c
```

$$\text{true}$$

$$\text{Type: Boolean}$$

Navigate the tree

```
first c
```

$$x$$

$$\text{Type: Symbol}$$

```
rest c
```

$$[x, [y, y]]$$

$$\text{Type: Magma Symbol}$$

```
rest rest c
```

$$[y, y]$$

<div align="right">Type: Magma Symbol</div>

Check ordering

```
ax:tree := a*x
```

$$[[x, x], x]$$

<div align="right">Type: Magma Symbol</div>

```
xa:tree := x*a
```

$$[x, [x, x]]$$

<div align="right">Type: Magma Symbol</div>

```
xa < ax
```

<div align="center">true</div>

<div align="right">Type: Boolean</div>

```
lexico(xa,ax)
```

<div align="center">false</div>

<div align="right">Type: Boolean</div>

9.50   MakeFunction

It is sometimes useful to be able to define a function given by the result of a calculation.

Suppose that you have obtained the following expression after several computations and that you now want to tabulate the numerical values of f for x between -1 and +1 with increment 0.1.

```
expr := (x - exp x + 1)**2 * (sin(x**2) * x + 1)**3
```

$$\left(x^3\ e^{x\,2} + \left(-2\ x^4 - 2\ x^3\right)\ e^x + x^5 + 2\ x^4 + x^3\right) \sin\left(x^2\right)^3 +$$

$$\left(3\ x^2\ e^{x\,2} + \left(-6\ x^3 - 6\ x^2\right)\ e^x + 3\ x^4 + 6\ x^3 + 3\ x^2\right) \sin\left(x^2\right)^2 +$$

$$\left(3\ x\ e^{x\,2} + \left(-6\ x^2 - 6\ x\right)\ e^x + 3\ x^3 + 6\ x^2 + 3\ x\right) \sin\left(x^2\right) + e^{x\,2} +$$

$$(-2\ x - 2)\ e^x + x^2 + 2\ x + 1$$

                                                            Type: Expression Integer

You could, of course, use the function **eval** within a loop and evaluate `expr`
twenty-one times, but this would be quite slow. A better way is to create a
numerical function `f` such that `f(x)` is defined by the expression `expr` above,
but without retyping `expr`! The package `MakeFunction` provides the operation
function which does exactly this.

Issue this to create the function `f(x)` given by `expr`.

```
function(expr, f, x)
```

$$f$$

                                                            Type: Symbol

To tabulate `expr`, we can now quickly evaluate `f` 21 times.

```
tbl := [f(0.1 * i - 1) for i in 0..20];
```

> [0.0005391844 0362701574, 0.0039657551 1844206653,
> 0.0088545187 4833983689 2, 0.0116524883 0907069695,
> 0.0108618220 9245751364 5, 0.0076366823 2120869965 06,
> 0.0040584985 7597822062 55, 0.0015349542 8910500836 48,
> 0.0003424903 1549879905 716, 0.0000233304 8276098819 6001,
> 0.0, 0.0000268186 8782862599 4229,
> 0.0004691571 3720051642 621, 0.0026924576 5968519586 08,
> 0.0101486881 7369135148 8, 0.0313833725 8543810564 3,
> 0.0876991144 5154615297 9, 0.2313019789 3439968362,
> 0.5843743955 958098772, 1.4114930171 992819197,
> 3.2216948276 75164252]

                                                            Type: List Float

Use the list `[x1,...,xn]` as the third argument to **function** to create a multi-
variate function `f(x1,...,xn)`.

```
e := (x - y + 1)**2 * (x**2 * y + 1)**2
```

$$x^4\ y^4 + \left(-2\ x^5 - 2\ x^4 + 2\ x^2\right)\ y^3 + \left(x^6 + 2\ x^5 + x^4 - 4\ x^3 - 4\ x^2 + 1\right)\ y^2 +$$

$$\left(2\ x^4 + 4\ x^3 + 2\ x^2 - 2\ x - 2\right)\ y + x^2 + 2\ x + 1$$

<div align="right">Type: Polynomial Integer</div>

```
function(e, g, [x, y])
```

$$g$$

<div align="right">Type: Symbol</div>

In the case of just two variables, they can be given as arguments without making them into a list.

```
function(e, h, x, y)
```

$$h$$

<div align="right">Type: Symbol</div>

Note that the functions created by **function** are not limited to floating point numbers, but can be applied to any type for which they are defined.

```
m1 := squareMatrix [ [1, 2], [3, 4] ]
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

<div align="right">Type: SquareMatrix(2,Integer)</div>

```
m2 := squareMatrix [ [1, 0], [-1, 1] ]
```

$$\begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$$

<div align="right">Type: SquareMatrix(2,Integer)</div>

```
h(m1, m2)
```

$$\begin{bmatrix} -7836 & 8960 \\ -17132 & 19588 \end{bmatrix}$$

<div align="right">Type: SquareMatrix(2,Integer)</div>

For more information, see **??** on page **??** in Section **??** on page **??**.

9.51   MappingPackage1

Function are objects of type `Mapping`. In this section we demonstrate some library operations from the packages `MappingPackage1`, `MappingPackage2`, and `MappingPackage3` that manipulate and create functions. Some terminology: a *nullary* function takes no arguments, a *unary* function takes one argument, and a *binary* function takes two arguments.

We begin by creating an example function that raises a rational number to an integer exponent.

```
power(q:  FRAC INT, n:  INT): FRAC INT == q**n
```

```
Function declaration power : (Fraction Integer,Integer) ->
   Fraction Integer has been added to workspace.
```

                                                                 Void

```
power(2,3)
```

```
Compiling function power with type (Fraction Integer,Integer) ->
   Fraction Integer
```

                                          8

                                              Type: Fraction Integer

The **twist** operation transposes the arguments of a binary function. Here `rewop(a, b)` is `power(b, a)`.

```
rewop := twist power
```

                                     theMap(...)

            Type: ((Integer,Fraction Integer) -> Fraction Integer)

This is 2^3.

```
rewop(3, 2)
```

                                          8

```
                                                Type:   Fraction Integer
```

Now we define `square` in terms of `power`.

```
square:  FRAC INT -> FRAC INT
```

```
                                                                    Void
```

The **curryRight** operation creates a unary function from a binary one by providing a constant argument on the right.

```
square:= curryRight(power, 2)
```

```
                              theMap(...)
```

```
                Type:  (Fraction Integer -> Fraction Integer)
```

Likewise, the **curryLeft** operation provides a constant argument on the left.

```
square 4
```

```
                                16
```

```
                                                Type:   Fraction Integer
```

The **constantRight** operation creates (in a trivial way) a binary function from a unary one: `constantRight(f)` is the function g such that `g(a,b)= f(a)`.

```
squirrel:= constantRight(square)$MAPPKG3(FRAC INT,FRAC INT,FRAC
INT)
```

```
                              theMap(...)
```

```
  Type:  ((Fraction Integer,Fraction Integer) -> Fraction Integer)
```

Likewise, `constantLeft(f)` is the function g such that `g(a,b)= f(b)`.

```
squirrel(1/2, 1/3)
```

$$\frac{1}{4}$$

                                        Type: Fraction Integer

The **curry** operation makes a unary function nullary.

```
sixteen := curry(square, 4/1)
```

                              theMap(...)

                                   Type: (() -> Fraction Integer)

```
sixteen()
```

                                   16

                                        Type: Fraction Integer

The "*" operation constructs composed functions.

```
square2:=square*square
```

                              theMap(...)

                    Type: (Fraction Integer -> Fraction Integer)

```
square2 3
```

                                   81

                                        Type: Fraction Integer

Use the "**" operation to create functions that are **n**-fold iterations of other functions.

```
sc(x:  FRAC INT): FRAC INT == x + 1
```

```
Function declaration sc : Fraction Integer ->
   Fraction Integer has been added to workspace.
```

                                                       Void

This is a list of `Mapping` objects.

```
incfns := [sc**i for i in 0..10]
```

$$[\text{theMap}(...), \text{theMap}(...), \text{theMap}(...), \text{theMap}(...), \text{theMap}(...), \text{theMap}(...),$$
$$\text{theMap}(...), \text{theMap}(...), \text{theMap}(...), \text{theMap}(...), \text{theMap}(...)]$$

```
              Type:  List (Fraction Integer -> Fraction Integer)
```

This is a list of applications of those functions.

```
[f 4 for f in incfns]
```

$$[4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]$$

```
                              Type:  List Fraction Integer
```

Use the **recur** operation for recursion:
```
g := recur f means g(n,x) == f(n,f(n-1,...f(1,x))).
```

```
times(n:NNI, i:INT):INT == n*i
```

```
Function declaration times : (NonNegativeInteger,Integer) ->
   Integer has been added to workspace.
```

```
                                                       Void
```

```
r := recur(times)
```

$$\text{theMap}(...)$$

```
                Type:  ((NonNegativeInteger,Integer) -> Integer)
```

This is a factorial function.

```
fact := curryRight(r, 1)
```

$$\text{theMap}(...)$$

```
                Type:  (NonNegativeInteger -> Integer)
```

```
fact 4
```

$$24$$

<div align="right">Type: PositiveInteger</div>

Constructed functions can be used within other functions.

```
mto2ton(m, n) ==
  raiser := square**n
  raiser m
```

<div align="right">Void</div>

This is 3^{2^3}.

```
mto2ton(3, 3)
```

```
   Compiling function mto2ton with type (PositiveInteger,
       PositiveInteger) -> Fraction Integer
```

$$6561$$

<div align="right">Type: Fraction Integer</div>

Here `shiftfib` is a unary function that modifies its argument.

```
shiftfib(r: List INT) : INT ==
  t := r.1
  r.1 := r.2
  r.2 := r.2 + t
  t
```

```
Function declaration shiftfib : List Integer -> Integer
   has been added to workspace.
```

<div align="right">Void</div>

By currying over the argument we get a function with private state.

```
fibinit:  List INT := [0, 1]
```

$$[0, 1]$$

```
fibs := curry(shiftfib, fibinit)
```

$$\text{theMap}(...)$$

Type: (() -> Integer)

```
[fibs() for i in 0..30]
```

$[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597,$
$2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418,$
$317811, 514229, 832040]$

Type: List Integer

9.52 Matrix

The `Matrix` domain provides arithmetic operations on matrices and standard functions from linear algebra. This domain is similar to the `TwoDimensionalArray` domain, except that the entries for `Matrix` must belong to a `Ring`.

9.52.1 Creating Matrices

There are many ways to create a matrix from a collection of values or from existing matrices.

If the matrix has almost all items equal to the same value, use **new** to create a matrix filled with that value and then reset the entries that are different.

```
m :  Matrix(Integer) := new(3,3,0)
```

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Type: Matrix Integer

To change the entry in the second row, third column to 5, use **setelt**.

```
setelt(m,2,3,5)
```

$$5$$

<div align="right">Type: PositiveInteger</div>

An alternative syntax is to use assignment.

```
m(1,2) := 10
```

$$10$$

<div align="right">Type: PositiveInteger</div>

The matrix was *destructively modified.*

```
m
```

$$\begin{bmatrix} 0 & 10 & 0 \\ 0 & 0 & 5 \\ 0 & 0 & 0 \end{bmatrix}$$

<div align="right">Type: Matrix Integer</div>

If you already have the matrix entries as a list of lists, use **matrix**.

```
matrix [ [1,2,3,4],[0,9,8,7] ]
```

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 9 & 8 & 7 \end{bmatrix}$$

<div align="right">Type: Matrix Integer</div>

If the matrix is diagonal, use **diagonalMatrix**.

```
dm := diagonalMatrix [1,x**2,x**3,x**4,x**5]
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & x^2 & 0 & 0 & 0 \\ 0 & 0 & x^3 & 0 & 0 \\ 0 & 0 & 0 & x^4 & 0 \\ 0 & 0 & 0 & 0 & x^5 \end{bmatrix}$$

Use **setRow** and **setColumn** to change a row or column of a matrix.

```
setRow!(dm,5,vector [1,1,1,1,1])
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & x^2 & 0 & 0 & 0 \\ 0 & 0 & x^3 & 0 & 0 \\ 0 & 0 & 0 & x^4 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Type: Matrix Polynomial Integer

```
setColumn!(dm,2,vector [y,y,y,y,y])
```

$$\begin{bmatrix} 1 & y & 0 & 0 & 0 \\ 0 & y & 0 & 0 & 0 \\ 0 & y & x^3 & 0 & 0 \\ 0 & y & 0 & x^4 & 0 \\ 1 & y & 1 & 1 & 1 \end{bmatrix}$$

Type: Matrix Polynomial Integer

Use **copy** to make a copy of a matrix.

```
cdm := copy(dm)
```

$$\begin{bmatrix} 1 & y & 0 & 0 & 0 \\ 0 & y & 0 & 0 & 0 \\ 0 & y & x^3 & 0 & 0 \\ 0 & y & 0 & x^4 & 0 \\ 1 & y & 1 & 1 & 1 \end{bmatrix}$$

Type: Matrix Polynomial Integer

This is useful if you intend to modify a matrix destructively but want a copy of the original.

```
setelt(dm,4,1,1-x**7)
```

$$-x^7 + 1$$

<div align="right">Type: Polynomial Integer</div>

`[dm,cdm]`

$$\left[\left[\begin{array}{ccccc} 1 & y & 0 & 0 & 0 \\ 0 & y & 0 & 0 & 0 \\ 0 & y & x^3 & 0 & 0 \\ -x^7+1 & y & 0 & x^4 & 0 \\ 1 & y & 1 & 1 & 1 \end{array}\right], \left[\begin{array}{ccccc} 1 & y & 0 & 0 & 0 \\ 0 & y & 0 & 0 & 0 \\ 0 & y & x^3 & 0 & 0 \\ 0 & y & 0 & x^4 & 0 \\ 1 & y & 1 & 1 & 1 \end{array}\right]\right]$$

<div align="right">Type: List Matrix Polynomial Integer</div>

Use **subMatrix** to extract part of an existing matrix. The syntax is subMatrix(*m, firstrow, lastrow, firstcol, lastcol*).

`subMatrix(dm,2,3,2,4)`

$$\left[\begin{array}{ccc} y & 0 & 0 \\ y & x^3 & 0 \end{array}\right]$$

<div align="right">Type: Matrix Polynomial Integer</div>

To change a submatrix, use **setsubMatrix**.

`d := diagonalMatrix [1.2,-1.3,1.4,-1.5]`

$$\left[\begin{array}{cccc} 1.2 & 0.0 & 0.0 & 0.0 \\ 0.0 & -1.3 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.4 & 0.0 \\ 0.0 & 0.0 & 0.0 & -1.5 \end{array}\right]$$

<div align="right">Type: Matrix Float</div>

If **e** is too big to fit where you specify, an error message is displayed. Use **subMatrix** to extract part of **e**, if necessary.

`e := matrix [ [6.7,9.11],[-31.33,67.19] ]`

$$\left[\begin{array}{cc} 6.7 & 9.11 \\ -31.33 & 67.19 \end{array}\right]$$

<div align="right">Type: Matrix Float</div>

This changes the submatrix of `d` whose upper left corner is at the first row and second column and whose size is that of `e`.

```
setsubMatrix!(d,1,2,e)
```

$$
\begin{bmatrix}
1.2 & 6.7 & 9.11 & 0.0 \\
0.0 & -31.33 & 67.19 & 0.0 \\
0.0 & 0.0 & 1.4 & 0.0 \\
0.0 & 0.0 & 0.0 & -1.5
\end{bmatrix}
$$

<div align="right">Type: Matrix Float</div>

`d`

$$
\begin{bmatrix}
1.2 & 6.7 & 9.11 & 0.0 \\
0.0 & -31.33 & 67.19 & 0.0 \\
0.0 & 0.0 & 1.4 & 0.0 \\
0.0 & 0.0 & 0.0 & -1.5
\end{bmatrix}
$$

<div align="right">Type: Matrix Float</div>

Matrices can be joined either horizontally or vertically to make new matrices.

```
a := matrix [ [1/2,1/3,1/4],[1/5,1/6,1/7] ]
```

$$
\begin{bmatrix}
\frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\
\frac{1}{5} & \frac{1}{6} & \frac{1}{7}
\end{bmatrix}
$$

<div align="right">Type: Matrix Fraction Integer</div>

```
b := matrix [ [3/5,3/7,3/11],[3/13,3/17,3/19] ]
```

$$
\begin{bmatrix}
\frac{3}{5} & \frac{3}{7} & \frac{3}{11} \\
\frac{3}{13} & \frac{3}{17} & \frac{3}{19}
\end{bmatrix}
$$

<div align="right">Type: Matrix Fraction Integer</div>

Use **horizConcat** to append them side to side. The two matrices must have the same number of rows.

```
horizConcat(a,b)
```

$$\left[\begin{array}{cccccc} \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{3}{5} & \frac{3}{7} & \frac{3}{11} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{3}{13} & \frac{3}{17} & \frac{3}{19} \end{array}\right]$$

<div align="right">Type: Matrix Fraction Integer</div>

Use **vertConcat** to stack one upon the other. The two matrices must have the same number of columns.

```
vab := vertConcat(a,b)
```

$$\left[\begin{array}{ccc} \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{3}{5} & \frac{3}{7} & \frac{3}{11} \\ \frac{3}{13} & \frac{3}{17} & \frac{3}{19} \end{array}\right]$$

<div align="right">Type: Matrix Fraction Integer</div>

The operation **transpose** is used to create a new matrix by reflection across the main diagonal.

```
transpose vab
```

$$\left[\begin{array}{cccc} \frac{1}{2} & \frac{1}{5} & \frac{3}{5} & \frac{3}{13} \\ \frac{1}{3} & \frac{1}{6} & \frac{3}{7} & \frac{3}{17} \\ \frac{1}{4} & \frac{1}{7} & \frac{3}{11} & \frac{3}{19} \end{array}\right]$$

<div align="right">Type: Matrix Fraction Integer</div>

9.52.2   Operations on Matrices

Axiom provides both left and right scalar multiplication.

```
m := matrix [ [1,2],[3,4] ]
```

$$\left[\begin{array}{cc} 1 & 2 \\ 3 & 4 \end{array}\right]$$

<div align="right">Type: Matrix Integer</div>

```
4 * m * (-5)
```

$$\left[\begin{array}{cc} -20 & -40 \\ -60 & -80 \end{array}\right]$$

You can add, subtract, and multiply matrices provided, of course, that the matrices have compatible dimensions. If not, an error message is displayed.

```
n := matrix([ [1,0,-2],[-3,5,1] ])
```

$$\begin{bmatrix} 1 & 0 & -2 \\ -3 & 5 & 1 \end{bmatrix}$$

Type: Matrix Integer

This following product is defined but `n * m` is not.

```
m * n
```

$$\begin{bmatrix} -5 & 10 & 0 \\ -9 & 20 & -2 \end{bmatrix}$$

Type: Matrix Integer

The operations **nrows** and **ncols** return the number of rows and columns of a matrix. You can extract a row or a column of a matrix using the operations **row** and **column**. The object returned is a `Vector`.

Here is the third column of the matrix `n`.

```
vec := column(n,3)
```

$$[-2, 1]$$

Type: Vector Integer

You can multiply a matrix on the left by a "row vector" and on the right by a "column vector."

```
vec * m
```

$$[1, 0]$$

Type: Vector Integer

Of course, the dimensions of the vector and the matrix must be compatible or an error message is returned.

```
m * vec
```

$$[0, -2]$$

<div align="right">Type: Vector Integer</div>

The operation **inverse** computes the inverse of a matrix if the matrix is invertible, and returns `"failed"` if not.

This Hilbert matrix is invertible.

```
hilb := matrix([ [1/(i + j) for i in 1..3] for j in 1..3])
```

$$\begin{bmatrix} \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \end{bmatrix}$$

<div align="right">Type: Matrix Fraction Integer</div>

```
inverse(hilb)
```

$$\begin{bmatrix} 72 & -240 & 180 \\ -240 & 900 & -720 \\ 180 & -720 & 600 \end{bmatrix}$$

<div align="right">Type: Union(Matrix Fraction Integer,...)</div>

This matrix is not invertible.

```
mm := matrix([ [1,2,3,4], [5,6,7,8], [9,10,11,12], [13,14,15,16]
])
```

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

<div align="right">Type: Matrix Integer</div>

```
inverse(mm)
```

$$\text{"failed"}$$

The operation **determinant** computes the determinant of a matrix provided that the entries of the matrix belong to a `CommutativeRing`.

The above matrix `mm` is not invertible and, hence, must have determinant 0.

```
determinant(mm)
```

$$0$$

Type: NonNegativeInteger

The operation **trace** computes the trace of a *square* matrix.

```
trace(mm)
```

$$34$$

Type: PositiveInteger

The operation **rank** computes the *rank* of a matrix: the maximal number of linearly independent rows or columns.

```
rank(mm)
```

$$2$$

Type: PositiveInteger

The operation **nullity** computes the *nullity* of a matrix: the dimension of its null space.

```
nullity(mm)
```

$$2$$

Type: PositiveInteger

The operation **nullSpace** returns a list containing a basis for the null space of a matrix. Note that the nullity is the number of elements in a basis for the null space.

`nullSpace(mm)`

$$[[1, -2, 1, 0], [2, -3, 0, 1]]$$

<div align="right">Type: List Vector Integer</div>

The operation **rowEchelon** returns the row echelon form of a matrix. It is easy to see that the rank of this matrix is two and that its nullity is also two.

`rowEchelon(mm)`

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 4 & 8 & 12 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

<div align="right">Type: Matrix Integer</div>

For more information on related topics, see **??** on page **??** in Section **??** on page **??**, **??** on page **??** in Section **??** on page **??**, **??** on page **??** in Section **??** on page **??**, 9.62 on page 733, 9.85 on page 837, 9.57 on page 715, and 9.82 on page 822.

9.53   MultiSet

The domain `Multiset(R)` is similar to `Set(R)` except that multiplicities (counts of duplications) are maintained and displayed. Use the operation **multiset** to create multisets from lists. All the standard operations from sets are available for multisets. An element with multiplicity greater than one has the multiplicity displayed first, then a colon, and then the element.

Create a multiset of integers.

`s := multiset [1,2,3,4,5,4,3,2,3,4,5,6,7,4,10]`

$$\{7, 2\colon 5, 3\colon 3, 1, 10, 6, 4\colon 4, 2\colon 2\}$$

<div align="right">Type: Multiset PositiveInteger</div>

The operation `insert!` adds an element to a multiset.

`insert!(3,s)`

$${7, 2\colon\ 5, 4\colon\ 3, 1, 10, 6, 4\colon\ 4, 2\colon\ 2}$$

<div align="right">Type: Multiset PositiveInteger</div>

Use `remove!` to remove an element. If a third argument is present, it specifies how many instances to remove. Otherwise all instances of the element are removed. Display the resulting multiset.

```
remove!(3,s,1); s
```

$${7, 2\colon\ 5, 3\colon\ 3, 1, 10, 6, 4\colon\ 4, 2\colon\ 2}$$

<div align="right">Type: Multiset PositiveInteger</div>

```
remove!(5,s); s
```

$${7, 3\colon\ 3, 1, 10, 6, 4\colon\ 4, 2\colon\ 2}$$

<div align="right">Type: Multiset PositiveInteger</div>

The operation `count` returns the number of copies of a given value.

```
count(5,s)
```

$$0$$

<div align="right">Type: NonNegativeInteger</div>

A second multiset.

```
t := multiset [2,2,2,-9]
```

$${-9, 3\colon\ 2}$$

<div align="right">Type: Multiset Integer</div>

The `union` of two multisets is additive.

```
U := union(s,t)
```

$${7, 3\colon\ 3, 1, -9, 10, 6, 4\colon\ 4, 5\colon\ 2}$$

<div align="right">Type: Multiset Integer</div>

The `intersect` operation gives the elements that are in common, with additive multiplicity.

```
I := intersect(s,t)
```

$$\{5:\ 2\}$$

<div align="right">Type: Multiset Integer</div>

The `difference` of `s` and `t` consists of the elements that `s` has but `t` does not. Elements are regarded as indistinguishable, so that if `s` and `t` have any element in common, the `difference` does not contain that element.

```
difference(s,t)
```

$$\{7, 3:\ 3, 1, 10, 6, 4:\ 4\}$$

<div align="right">Type: Multiset Integer</div>

The `symmetricDifference` is the union of `difference(s, t)` and `difference(t, s)`.

```
S := symmetricDifference(s,t)
```

$$\{7, 3:\ 3, 1, -9, 10, 6, 4:\ 4\}$$

<div align="right">Type: Multiset Integer</div>

Check that the `union` of the `symmetricDifference` and the `intersect` equals the `union` of the elements.

```
(U = union(S,I))@Boolean
```

$$true$$

<div align="right">Type: Boolean</div>

Check some inclusion relations.

```
t1 := multiset [1,2,2,3]; [t1 < t, t1 < s, t < s, t1 <= s]
```

$$[\mathtt{false}, \mathtt{true}, \mathtt{false}, \mathtt{true}]$$

<div align="right">Type: List Boolean</div>

9.54 MultivariatePolynomial

The domain constructor `MultivariatePolynomial` is similar to `Polynomial` except that it specifies the variables to be used. `Polynomial` are available for `MultivariatePolynomial`. The abbreviation for `MultivariatePolynomial` is `MPOLY`. The type expressions

$$\text{MultivariatePolynomial([x,y],Integer)}$$

and

$$\text{MPOLY([x,y],INT)}$$

refer to the domain of multivariate polynomials in the variables x and y where the coefficients are restricted to be integers. The first variable specified is the main variable and the display of the polynomial reflects this.

This polynomial appears with terms in descending powers of the variable x.

```
m :  MPOLY([x,y],INT) := (x**2 - x*y**3 +3*y)**2
```

$$x^4 - 2\ y^3\ x^3 + \left(y^6 + 6\ y\right)\ x^2 - 6\ y^4\ x + 9\ y^2$$

<div align="center">Type: MultivariatePolynomial([x,y],Integer)</div>

It is easy to see a different variable ordering by doing a conversion.

```
m ::  MPOLY([y,x],INT)
```

$$x^2\ y^6 - 6\ x\ y^4 - 2\ x^3\ y^3 + 9\ y^2 + 6\ x^2\ y + x^4$$

<div align="center">Type: MultivariatePolynomial([y,x],Integer)</div>

You can use other, unspecified variables, by using `Polynomial` in the coefficient type of `MPOLY`.

```
p :  MPOLY([x,y],POLY INT)
```

<div align="right">Void</div>

```
p := (a**2*x - b*y**2 + 1)**2
```

$$a^4\ x^2 + \left(-2\ a^2\ b\ y^2 + 2\ a^2\right)\ x + b^2\ y^4 - 2\ b\ y^2 + 1$$

<div align="center">Type: MultivariatePolynomial([x,y],Polynomial Integer)</div>

Conversions can be used to re-express such polynomials in terms of the other variables. For example, you can first push all the variables into a polynomial with integer coefficients.

```
p ::  POLY INT
```

$$b^2 \, y^4 + \left(-2 \, a^2 \, b \, x - 2 \, b\right) y^2 + a^4 \, x^2 + 2 \, a^2 \, x + 1$$

<div align="right">Type: Polynomial Integer</div>

Now pull out the variables of interest.

```
% ::  MPOLY([a,b],POLY INT)
```

$$x^2 \, a^4 + \left(-2 \, x \, y^2 \, b + 2 \, x\right) a^2 + y^4 \, b^2 - 2 \, y^2 \, b + 1$$

<div align="center">Type: MultivariatePolynomial([a,b],Polynomial Integer)</div>

Restriction:

> Axiom does not allow you to create types where `MultivariatePolynomial` is contained in the coefficient type of `Polynomial`. Therefore, `MPOLY([x,y],POLY INT)` is legal but `POLY MPOLY([x,y],INT)` is not.

Multivariate polynomials may be combined with univariate polynomials to create types with special structures.

```
q :  UP(x, FRAC MPOLY([y,z],INT))
```

<div align="right">Void</div>

This is a polynomial in `x` whose coefficients are quotients of polynomials in `y` and `z`.

```
q := (x**2 - x*(z+1)/y +2)**2
```

$$x^4 + \frac{-2 \, z - 2}{y} \, x^3 + \frac{4 \, y^2 + z^2 + 2 \, z + 1}{y^2} \, x^2 + \frac{-4 \, z - 4}{y} \, x + 4$$

<div align="right">Type: UnivariatePolynomial(x,Fraction
MultivariatePolynomial([y,z],Integer))</div>

Use conversions for structural rearrangements. `z` does not appear in a denominator and so it can be made the main variable.

```
q ::  UP(z, FRAC MPOLY([x,y],INT))
```

$$\frac{x^2}{y^2}\, z^2 + \frac{-2\, y\, x^3 + 2\, x^2 - 4\, y\, x}{y^2}\, z + \frac{y^2\, x^4 - 2\, y\, x^3 + \left(4\, y^2 + 1\right) x^2 - 4\, y\, x + 4\, y^2}{y^2}$$

<div align="right">

Type: UnivariatePolynomial(z,Fraction
MultivariatePolynomial([x,y],Integer))

</div>

Or you can make a multivariate polynomial in x and z whose coefficients are fractions in polynomials in y.

```
q ::  MPOLY([x,z], FRAC UP(y,INT))
```

$$x^4 + \left(-\frac{2}{y}\, z - \frac{2}{y}\right) x^3 + \left(\frac{1}{y^2}\, z^2 + \frac{2}{y^2}\, z + \frac{4\, y^2 + 1}{y^2}\right) x^2 +$$

$$\left(-\frac{4}{y}\, z - \frac{4}{y}\right) x + 4$$

<div align="right">

Type: MultivariatePolynomial([x,z],Fraction
UnivariatePolynomial(y,Integer))

</div>

A conversion like `q ::  MPOLY([x,y], FRAC UP(z,INT))` is not possible in this example because y appears in the denominator of a fraction. As you can see, Axiom provides extraordinary flexibility in the manipulation and display of expressions via its conversion facility.

For more information on related topics, see 9.63 on page 734, 9.83 on page 827, and 9.16 on page 531.

9.55   None

The `None` domain is not very useful for interactive work but it is provided nevertheless for completeness of the Axiom type system.

Probably the only place you will ever see it is if you enter an empty list with no type information.

```
[ ]
```

<div align="center">

[ ]

</div>

<div align="right">

Type: List None

</div>

Such an empty list can be converted into an empty list of any other type.

```
[ ] ::  List Float
```

$$[\,]$$

<div align="right">Type: List Float</div>

If you wish to produce an empty list of a particular type directly, such as `List NonNegativeInteger`, do it this way.

```
[ ]$List(NonNegativeInteger)
```

$$[\,]$$

<div align="right">Type: List NonNegativeInteger</div>

9.56   Octonion

The Octonions, also called the Cayley-Dixon algebra, defined over a commutative ring are an eight-dimensional non-associative algebra. Their construction from quaternions is similar to the construction of quaternions from complex numbers (see 9.64 on page 745).

As `Octonion` creates an eight-dimensional algebra, you have to give eight components to construct an octonion.

```
oci1 := octon(1,2,3,4,5,6,7,8)
```

$$1 + 2\ i + 3\ j + 4\ k + 5\ E + 6\ I + 7\ J + 8\ K$$

<div align="right">Type: Octonion Integer</div>

```
oci2 := octon(7,2,3,-4,5,6,-7,0)
```

$$7 + 2\ i + 3\ j - 4\ k + 5\ E + 6\ I - 7\ J$$

<div align="right">Type: Octonion Integer</div>

Or you can use two quaternions to create an octonion.

```
oci3 := octon(quatern(-7,-12,3,-10), quatern(5,6,9,0))
```

$$-7 - 12\ i + 3\ j - 10\ k + 5\ E + 6\ I + 9\ J$$

<div align="right">Type: Octonion Integer</div>

You can easily demonstrate the non-associativity of multiplication.

```
(oci1 * oci2) * oci3 - oci1 * (oci2 * oci3)
```

$$2696\ i - 2928\ j - 4072\ k + 16\ E - 1192\ I + 832\ J + 2616\ K$$

<div align="right">Type: Octonion Integer</div>

As with the quaternions, we have a real part, the imaginary parts i, j, k, and four additional imaginary parts E, I, J and K. These parts correspond to the canonical basis (1,i,j,k,E,I,J,K).

For each basis element there is a component operation to extract the coefficient of the basis element for a given octonion.

```
[real oci1, imagi oci1, imagj oci1, imagk oci1, imagE oci1, imagI
oci1, imagJ oci1, imagK oci1]
```

$$[1, 2, 3, 4, 5, 6, 7, 8]$$

<div align="right">Type: List PositiveInteger</div>

A basis with respect to the quaternions is given by (1,E). However, you might ask, what then are the commuting rules? To answer this, we create some generic elements.

We do this in Axiom by simply changing the ground ring from Integer to Polynomial Integer.

```
q :  Quaternion Polynomial Integer := quatern(q1, qi, qj, qk)
```

$$q1 + qi\ i + qj\ j + qk\ k$$

<div align="right">Type: Quaternion Polynomial Integer</div>

```
E : Octonion Polynomial Integer:= octon(0,0,0,0,1,0,0,0)
```

$$E$$

<div align="right">Type: Octonion Polynomial Integer</div>

Note that quaternions are automatically converted to octonions in the obvious way.

q * E

$$q1\ E + qi\ I + qj\ J + qk\ K$$

<div align="right">Type: Octonion Polynomial Integer</div>

E * q

$$q1\ E - qi\ I - qj\ J - qk\ K$$

<div align="right">Type: Octonion Polynomial Integer</div>

q * 1$(Octonion Polynomial Integer)

$$q1 + qi\ i + qj\ j + qk\ k$$

<div align="right">Type: Octonion Polynomial Integer</div>

1$(Octonion Polynomial Integer) * q

$$q1 + qi\ i + qj\ j + qk\ k$$

<div align="right">Type: Octonion Polynomial Integer</div>

Finally, we check that the **norm**, defined as the sum of the squares of the coefficients, is a multiplicative map.

o :   Octonion Polynomial Integer := octon(o1, oi, oj, ok, oE, oI, oJ, oK)

$$o1 + oi\ i + oj\ j + ok\ k + oE\ E + oI\ I + oJ\ J + oK\ K$$

<div align="right">Type: Octonion Polynomial Integer</div>

norm o

$$ok^2 + oj^2 + oi^2 + oK^2 + oJ^2 + oI^2 + oE^2 + o1^2$$

                                         Type: Polynomial Integer

```
p :  Octonion Polynomial Integer := octon(p1, pi, pj, pk, pE, pI,
pJ, pK)
```

$$p1 + pi\ i + pj\ j + pk\ k + pE\ E + pI\ I + pJ\ J + pK\ K$$

                              Type: Octonion Polynomial Integer

Since the result is 0, the norm is multiplicative.

```
norm(o*p)-norm(p)*norm(o)
```

$$0$$

                                         Type: Polynomial Integer

9.57 OneDimensionalArray

The `OneDimensionalArray` domain is used for storing data in a one-dimensional indexed data structure. Such an array is a homogeneous data structure in that all the entries of the array must belong to the same Axiom domain. Each array has a fixed length specified by the user and arrays are not extensible. The indexing of one-dimensional arrays is one-based. This means that the "first" element of an array is given the index 1. See also 9.85 on page 837 and 9.26 on page 561.

To create a one-dimensional array, apply the operation `oneDimensionalArray` to a list.

```
oneDimensionalArray [i**2 for i in 1..10]
```

$$[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]$$

                              Type: OneDimensionalArray PositiveInteger

Another approach is to first create `a`, a one-dimensional array of 10 0's. `OneDimensionalArray` has the convenient abbreviation `ARRAY1`.

```
a :  ARRAY1 INT := new(10,0)
```

$$[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$$

                                    Type: OneDimensionalArray Integer

Set each `i`th element to i, then display the result.

```
for i in 1..10 repeat a.i := i; a
```

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

                                    Type: OneDimensionalArray Integer

Square each element by mapping the function $i \mapsto i^2$ onto each element.

```
map!(i +-> i ** 2,a); a
```

$$[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]$$

                                    Type: OneDimensionalArray Integer

Reverse the elements in place.

```
reverse!  a
```

$$[100, 81, 64, 49, 36, 25, 16, 9, 4, 1]$$

                                    Type: OneDimensionalArray Integer

Swap the 4th and 5th element.

```
swap!(a,4,5); a
```

$$[100, 81, 64, 36, 49, 25, 16, 9, 4, 1]$$

                                    Type: OneDimensionalArray Integer

Sort the elements in place.

```
sort!  a
```

$$[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]$$

                                    Type: OneDimensionalArray Integer

Create a new one-dimensional array `b` containing the last 5 elements of `a`.

```
b := a(6..10)
```

$$[36, 49, 64, 81, 100]$$

<div align="right">Type: OneDimensionalArray Integer</div>

Replace the first 5 elements of `a` with those of `b`.

```
copyInto!(a,b,1)
```

$$[36, 49, 64, 81, 100, 36, 49, 64, 81, 100]$$

<div align="right">Type: OneDimensionalArray Integer</div>

9.58 Operator

Given any ring `R`, the ring of the `Integer`-linear operators over `R` is called `Operator(R)`. To create an operator over `R`, first create a basic operator using the operation `operator`, and then convert it to `Operator(R)` for the `R` you want.

We choose `R` to be the two by two matrices over the integers.

```
R := SQMATRIX(2, INT)
```

$$SquareMatrix(2, Integer)$$

<div align="right">Type: Domain</div>

Create the operator `tilde` on `R`.

```
t := operator("tilde") ::  OP(R)
```

$$tilde$$

<div align="right">Type: Operator SquareMatrix(2,Integer)</div>

Since `Operator` is unexposed we must either package-call operations from it, or expose it explicitly. For convenience we will do the latter.

Expose `Operator`.

```
)set expose add constructor Operator
```

   Operator is now explicitly exposed in frame G82322

To attach an evaluation function (from `R` to `R`) to an operator over `R`, use
`evaluate(op, f)` where `op` is an operator over `R` and `f` is a function `R ->`
`R`. This needs to be done only once when the operator is defined. Note that `f`
must be `Integer`-linear (that is, `f(ax+y) = a f(x) + f(y)` for any integer `a`,
and any `x` and `y` in `R`).

We now attach the transpose map to the above operator `t`.

```
evaluate(t, m +-> transpose m)
```

tilde

                           Type: Operator SquareMatrix(2,Integer)

Operators can be manipulated formally as in any ring: `+` is the pointwise addi-
tion and `*` is composition. Any element `x` of `R` can be converted to an operator
op_x over `R`, and the evaluation function of op_x is left-multiplication by `x`.

Multiplying on the left by this matrix swaps the two rows.

```
s :  R := matrix [ [0, 1], [1, 0] ]
```

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

                           Type: SquareMatrix(2,Integer)

Can you guess what is the action of the following operator?

```
rho := t * s
```

$$tilde \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

                           Type: Operator SquareMatrix(2,Integer)

Hint: applying `rho` four times gives the identity, so `rho**4-1` should return 0
when applied to any two by two matrix.

```
z := rho**4 - 1
```

$$-1 + tilde \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} tilde \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} tilde \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} tilde \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

                          Type: Operator SquareMatrix(2,Integer)

Now check with this matrix.

```
m:R := matrix [ [1, 2], [3, 4] ]
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

                          Type: SquareMatrix(2,Integer)

```
z m
```

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

                          Type: SquareMatrix(2,Integer)

As you have probably guessed by now, `rho` acts on matrices by rotating the elements clockwise.

```
rho m
```

$$\begin{bmatrix} 3 & 1 \\ 4 & 2 \end{bmatrix}$$

                          Type: SquareMatrix(2,Integer)

```
rho rho m
```

$$\begin{bmatrix} 4 & 3 \\ 2 & 1 \end{bmatrix}$$

                          Type: SquareMatrix(2,Integer)

```
(rho**3) m
```

$$\begin{bmatrix} 2 & 4 \\ 1 & 3 \end{bmatrix}$$

<div align="right">Type: SquareMatrix(2,Integer)</div>

Do the swapping of rows and transposition commute? We can check by computing their bracket.

```
b := t * s - s * t
```

$$-\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} tilde + tilde \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

<div align="right">Type: Operator SquareMatrix(2,Integer)</div>

Now apply it to m.

```
b m
```

$$\begin{bmatrix} 1 & -3 \\ 3 & -1 \end{bmatrix}$$

<div align="right">Type: SquareMatrix(2,Integer)</div>

Next we demonstrate how to define a differential operator on a polynomial ring. This is the recursive definition of the n-th Legendre polynomial.

```
L n ==
  n = 0 => 1
  n = 1 => x
  (2*n-1)/n * x * L(n-1) - (n-1)/n * L(n-2)
```

<div align="right">Void</div>

Create the differential operator $\frac{d}{dx}$ on polynomials in x over the rational numbers.

```
dx := operator("D") ::  OP(POLY FRAC INT)
```

$$D$$

<div align="right">Type: Operator Polynomial Fraction Integer</div>

Now attach the map to it.

```
evaluate(dx, p +-> D(p, 'x))
```

$$D$$

<div align="center">Type: Operator Polynomial Fraction Integer</div>

This is the differential equation satisfied by the **n**-th Legendre polynomial.

```
E n == (1 - x**2) * dx**2 - 2 * x * dx + n*(n+1)
```

<div align="right">Void</div>

Now we verify this for **n = 15**. Here is the polynomial.

```
L 15
```

$$\frac{9694845}{2048}\, x^{15} - \frac{35102025}{2048}\, x^{13} + \frac{50702925}{2048}\, x^{11} - \frac{37182145}{2048}\, x^9 + \frac{14549535}{2048}\, x^7 -$$

$$\frac{2909907}{2048}\, x^5 + \frac{255255}{2048}\, x^3 - \frac{6435}{2048}\, x$$

<div align="center">Type: Polynomial Fraction Integer</div>

Here is the operator.

```
E 15
```

$$240 - 2\, x\, D - \left(x^2 - 1\right) D^2$$

<div align="center">Type: Operator Polynomial Fraction Integer</div>

Here is the evaluation.

```
(E 15)(L 15)
```

$$0$$

<div align="center">Type: Polynomial Fraction Integer</div>

9.59   OrderedVariableList

The domain `OrderedVariableList` provides symbols which are restricted to a
particular list and have a definite ordering. Those two features are specified by
a `List Symbol` object that is the argument to the domain.

This is a sample ordering of three symbols.

```
ls:List Symbol:=['x,'a,'z]
```

$$[x, a, z]$$

<div align="right">Type: List Symbol</div>

Let's build the domain

```
Z:=OVAR ls
```

<div align="center">OrderedVariableList [x,a,z]</div>

<div align="right">Type: Domain</div>

How many variables does it have?

```
size()$Z
```

<div align="center">3</div>

<div align="right">Type: NonNegativeInteger</div>

They are (in the imposed order)

```
lv:=[index(i::PI)$Z for i in 1..size()$Z]
```

$$[x, a, z]$$

<div align="right">Type: List OrderedVariableList [x,a,z]</div>

Check that the ordering is right

```
sorted?(>,lv)
```

<div align="center">true</div>

<div align="right">Type: Boolean</div>

9.60 OrderlyDifferentialPolynomial

Many systems of differential equations may be transformed to equivalent systems of ordinary differential equations where the equations are expressed polynomially in terms of the unknown functions. In Axiom, the domain constructors `OrderlyDifferentialPolynomial` (abbreviated `ODPOL`) and `SequentialDifferentialPolynomial` (abbreviation `SDPOL`) implement two domains of ordinary differential polynomials over any differential ring. In the simplest case, this differential ring is usually either the ring of integers, or the field of rational numbers. However, Axiom can handle ordinary differential polynomials over a field of rational functions in a single indeterminate.

The two domains `ODPOL` and `SDPOL` are almost identical, the only difference being the choice of a different ranking, which is an ordering of the derivatives of the indeterminates. The first domain uses an orderly ranking, that is, derivatives of higher order are ranked higher, and derivatives of the same order are ranked alphabetically. The second domain uses a sequential ranking, where derivatives are ordered first alphabetically by the differential indeterminates, and then by order. A more general domain constructor, `DifferentialSparseMultivariatePolynomial` (abbreviation `DSMP`) allows both a user-provided list of differential indeterminates as well as a user-defined ranking. We shall illustrate `ODPOL(FRAC INT)`, which constructs a domain of ordinary differential polynomials in an arbitrary number of differential indeterminates with rational numbers as coefficients.

```
dpol:= ODPOL(FRAC INT)
```

> OrderlyDifferentialPolynomial Fraction Integer

> Type: Domain

A differential indeterminate `w` may be viewed as an infinite sequence of algebraic indeterminates, which are the derivatives of `w`. To facilitate referencing these, Axiom provides the operation **makeVariable** to convert an element of type `Symbol` to a map from the natural numbers to the differential polynomial ring.

```
w := makeVariable('w)$dpol
```

> theMap(...)

> Type: (NonNegativeInteger -> OrderlyDifferentialPolynomial
> Fraction Integer)

```
z := makeVariable('z)$dpol
```

$$\text{theMap}(...)$$

> Type: (NonNegativeInteger -> OrderlyDifferentialPolynomial
>   Fraction Integer)

The fifth derivative of `w` can be obtained by applying the map `w` to the number
`5`. Note that the order of differentiation is given as a subscript (except when
the order is 0).

`w.5`

$$w_5$$

> Type: OrderlyDifferentialPolynomial Fraction Integer

`w 0`

$$w$$

> Type: OrderlyDifferentialPolynomial Fraction Integer

The first five derivatives of `z` can be generated by a list.

`[z.i for i in 1..5]`

$$[z_1, z_2, z_3, z_4, z_5]$$

> Type: List OrderlyDifferentialPolynomial Fraction Integer

The usual arithmetic can be used to form a differential polynomial from the
derivatives.

`f:= w.4 - w.1 * w.1 * z.3`

$$w_4 - w_1{}^2\, z_3$$

> Type: OrderlyDifferentialPolynomial Fraction Integer

`g:=(z.1)**3 * (z.2)**2 - w.2`

$$z_1{}^3\, z_2{}^2 - w_2$$

$$\text{Type:}\quad \texttt{OrderlyDifferentialPolynomial Fraction Integer}$$

The operation **D** computes the derivative of any differential polynomial.

```
D(f)
```

$$w_5 - {w_1}^2\ z_4 - 2\ w_1\ w_2\ z_3$$

$$\text{Type:}\quad \texttt{OrderlyDifferentialPolynomial Fraction Integer}$$

The same operation can compute higher derivatives, like the fourth derivative.

```
D(f,4)
```

$$w_8 - {w_1}^2\ z_7 - 8\ w_1\ w_2\ z_6 + \left(-12\ w_1\ w_3 - 12\ {w_2}^2\right)\ z_5 - 2\ w_1\ z_3\ w_5 +$$

$$\left(-8\ w_1\ w_4 - 24\ w_2\ w_3\right)\ z_4 - 8\ w_2\ z_3\ w_4 - 6\ {w_3}^2\ z_3$$

$$\text{Type:}\quad \texttt{OrderlyDifferentialPolynomial Fraction Integer}$$

The operation **makeVariable** creates a map to facilitate referencing the derivatives of f, similar to the map w.

```
df:=makeVariable(f)$dpol
```

$$\text{theMap}(...)$$

$$\text{Type:}\quad \texttt{(NonNegativeInteger -> OrderlyDifferentialPolynomial}$$
$$\texttt{Fraction Integer)}$$

The fourth derivative of f may be referenced easily.

```
df.4
```

$$w_8 - {w_1}^2\ z_7 - 8\ w_1\ w_2\ z_6 + \left(-12\ w_1\ w_3 - 12\ {w_2}^2\right)\ z_5 - 2\ w_1\ z_3\ w_5 +$$

$$\left(-8\ w_1\ w_4 - 24\ w_2\ w_3\right)\ z_4 - 8\ w_2\ z_3\ w_4 - 6\ {w_3}^2\ z_3$$

$$\text{Type:}\quad \texttt{OrderlyDifferentialPolynomial Fraction Integer}$$

The operation **order** returns the order of a differential polynomial, or the order in a specified differential indeterminate.

```
order(g)
```

$$2$$

                                        Type: PositiveInteger

```
order(g, 'w)
```

$$2$$

                                        Type: PositiveInteger

The operation **differentialVariables** returns a list of differential indeterminates occurring in a differential polynomial.

```
differentialVariables(g)
```

$$[z, w]$$

                                          Type: List Symbol

The operation **degree** returns the degree, or the degree in the differential indeterminate specified.

```
degree(g)
```

$${z_2}^2 \, {z_1}^3$$

        Type: IndexedExponents OrderlyDifferentialVariable Symbol

```
degree(g, 'w)
```

$$1$$

                                          Type: PositiveInteger

The operation **weights** returns a list of weights of differential monomials appearing in differential polynomial, or a list of weights in a specified differential indeterminate.

```
weights(g)
```

$$[7, 2]$$

<div align="right">Type: List NonNegativeInteger</div>

weights(g,'w)

$$[2]$$

<div align="right">Type: List NonNegativeInteger</div>

The operation **weight** returns the maximum weight of all differential monomials appearing in the differential polynomial.

weight(g)

$$7$$

<div align="right">Type: PositiveInteger</div>

A differential polynomial is *isobaric* if the weights of all differential monomials appearing in it are equal.

isobaric?(g)

<div align="center">false</div>

<div align="right">Type: Boolean</div>

To substitute *differentially*, use **eval**. Note that we must coerce 'w to Symbol, since in ODPOL, differential indeterminates belong to the domain Symbol. Compare this result to the next, which substitutes *algebraically* (no substitution is done since w.0 does not appear in g).

eval(g,['w::Symbol],[f])

$$-w_6 + w_1{}^2 \ z_5 + 4 \ w_1 \ w_2 \ z_4 + \left(2 \ w_1 \ w_3 + 2 \ w_2{}^2\right) z_3 + z_1{}^3 \ z_2{}^2$$

<div align="right">Type: OrderlyDifferentialPolynomial Fraction Integer</div>

eval(g,variables(w.0),[f])

$$z_1{}^3 \ z_2{}^2 - w_2$$

Type: OrderlyDifferentialPolynomial Fraction Integer

Since `OrderlyDifferentialPolynomial` belongs to `PolynomialCategory`, all
the operations defined in the latter category, or in packages for the latter cate-
gory, are available.

`monomials(g)`

$$\left[{z_1}^3\ {z_2}^2, -w_2 \right]$$

Type: List OrderlyDifferentialPolynomial Fraction Integer

`variables(g)`

$$\left[z_2, w_2, z_1 \right]$$

Type: List OrderlyDifferentialVariable Symbol

`gcd(f,g)`

$$1$$

Type: OrderlyDifferentialPolynomial Fraction Integer

`groebner([f,g])`

$$\left[w_4 - {w_1}^2\ z_3, {z_1}^3\ {z_2}^2 - w_2 \right]$$

Type: List OrderlyDifferentialPolynomial Fraction Integer

The next three operations are essential for elimination procedures in differential
polynomial rings. The operation **leader** returns the leader of a differential poly-
nomial, which is the highest ranked derivative of the differential indeterminates
that occurs.

`lg:=leader(g)`

$$z_2$$

Type: OrderlyDifferentialVariable Symbol

The operation **separant** returns the separant of a differential polynomial, which is the partial derivative with respect to the leader.

sg:=separant(g)

$$2 \; {z_1}^3 \; z_2$$

Type: OrderlyDifferentialPolynomial Fraction Integer

The operation **initial** returns the initial, which is the leading coefficient when the given differential polynomial is expressed as a polynomial in the leader.

ig:=initial(g)

$${z_1}^3$$

Type: OrderlyDifferentialPolynomial Fraction Integer

Using these three operations, it is possible to reduce `f` modulo the differential ideal generated by `g`. The general scheme is to first reduce the order, then reduce the degree in the leader. First, eliminate `z.3` using the derivative of `g`.

g1 := D g

$$2 \; {z_1}^3 \; z_2 \; z_3 - w_3 + 3 \; {z_1}^2 \; {z_2}^3$$

Type: OrderlyDifferentialPolynomial Fraction Integer

Find its leader.

lg1:= leader g1

$$z_3$$

Type: OrderlyDifferentialVariable Symbol

Differentiate `f` partially with respect to this leader.

pdf:=D(f, lg1)

$$-{w_1}^2$$

<div align="center">Type: OrderlyDifferentialPolynomial Fraction Integer</div>

Compute the partial remainder of `f` with respect to `g`.

`prf:=sg * f- pdf * g1`

$$2 \; z_1{}^3 \; z_2 \; w_4 - w_1{}^2 \; w_3 + 3 \; w_1{}^2 \; z_1{}^2 \; z_2{}^3$$

<div align="center">Type: OrderlyDifferentialPolynomial Fraction Integer</div>

Note that high powers of `lg` still appear in `prf`. Compute the leading coefficient of `prf` as a polynomial in the leader of `g`.

`lcf:=leadingCoefficient univariate(prf, lg)`

$$3 \; w_1{}^2 \; z_1{}^2$$

<div align="center">Type: OrderlyDifferentialPolynomial Fraction Integer</div>

Finally, continue eliminating the high powers of `lg` appearing in `prf` to obtain the (pseudo) remainder of `f` modulo `g` and its derivatives.

`ig * prf - lcf * g * lg`

$$2 \; z_1{}^6 \; z_2 \; w_4 - w_1{}^2 \; z_1{}^3 \; w_3 + 3 \; w_1{}^2 \; z_1{}^2 \; w_2 \; z_2$$

<div align="center">Type: OrderlyDifferentialPolynomial Fraction Integer</div>

9.61   PartialFraction

A *partial fraction* is a decomposition of a quotient into a sum of quotients where the denominators of the summands are powers of primes.[5] For example, the rational number `1/6` is decomposed into `1/2-1/3`. You can compute partial fractions of quotients of objects from domains belonging to the category `EuclideanDomain`. For example, `Integer`, `Complex Integer`, and `UnivariatePolynomial(x, Fraction Integer)` all belong to `EuclideanDomain`. In the examples following, we demonstrate how to decompose quotients of each of these kinds of object into partial fractions. Issue the system command `)show PartialFraction` to display the full list of operations defined by `PartialFraction`.

---

[5]Most people first encounter partial fractions when they are learning integral calculus. For a technical discussion of partial fractions, see, for example, Lang's *Algebra.*

It is necessary that we know how to factor the denominator when we want to compute a partial fraction. Although the interpreter can often do this automatically, it may be necessary for you to include a call to `factor`. In these examples, it is not necessary to factor the denominators explicitly.

The main operation for computing partial fractions is called **partialFraction** and we use this to compute a decomposition of `1 / 10!`. The first argument to **partialFraction** is the numerator of the quotient and the second argument is the factored denominator.

```
partialFraction(1,factorial 10)
```

$$\frac{159}{2^8} - \frac{23}{3^4} - \frac{12}{5^2} + \frac{1}{7}$$

<div align="right">Type: PartialFraction Integer</div>

Since the denominators are powers of primes, it may be possible to expand the numerators further with respect to those primes. Use the operation **padicFraction** to do this.

```
f := padicFraction(%)
```

$$\frac{1}{2} + \frac{1}{2^4} + \frac{1}{2^5} + \frac{1}{2^6} + \frac{1}{2^7} + \frac{1}{2^8} - \frac{2}{3^2} - \frac{1}{3^3} - \frac{2}{3^4} - \frac{2}{5} - \frac{2}{5^2} + \frac{1}{7}$$

<div align="right">Type: PartialFraction Integer</div>

The operation **compactFraction** returns an expanded fraction into the usual form. The compacted version is used internally for computational efficiency.

```
compactFraction(f)
```

$$\frac{159}{2^8} - \frac{23}{3^4} - \frac{12}{5^2} + \frac{1}{7}$$

<div align="right">Type: PartialFraction Integer</div>

You can add, subtract, multiply and divide partial fractions. In addition, you can extract the parts of the decomposition. **numberOfFractionalTerms** computes the number of terms in the fractional part. This does not include the whole part of the fraction, which you get by calling **wholePart**. In this example, the whole part is just `0`.

```
numberOfFractionalTerms(f)
```

$$12$$

<div align="right">Type: PositiveInteger</div>

The operation **nthFractionalTerm** returns the individual terms in the decomposition. Notice that the object returned is a partial fraction itself. **firstNumer** and **firstDenom** extract the numerator and denominator of the first term of the fraction.

```
nthFractionalTerm(f,3)
```

$$\frac{1}{2^5}$$

<div align="right">Type: PartialFraction Integer</div>

Given two gaussian integers (see 9.11 on page 501), you can decompose their quotient into a partial fraction.

```
partialFraction(1,- 13 + 14 * %i)
```

$$-\frac{1}{1+2\ i} + \frac{4}{3+8\ i}$$

<div align="right">Type: PartialFraction Complex Integer</div>

To convert back to a quotient, simply use a conversion.

```
% ::  Fraction Complex Integer
```

$$-\frac{i}{14+13\ i}$$

<div align="right">Type: Fraction Complex Integer</div>

To conclude this section, we compute the decomposition of

```
                 1
     -------------------------------
                   2        3        4
      (x + 1)(x + 2)  (x + 3)  (x + 4)
```

The polynomials in this object have type `UnivariatePolynomial(x, Fraction Integer)`.

We use the **primeFactor** operation (see 9.22 on page 546) to create the denominator in factored form directly.

```
u :  FR UP(x, FRAC INT) := reduce(*,[primeFactor(x+i,i) for i in
1..4])
```

$$(x + 1) \, (x + 2)^2 \, (x + 3)^3 \, (x + 4)^4$$

Type: Factored UnivariatePolynomial(x,Fraction Integer)

These are the compact and expanded partial fractions for the quotient.

```
partialFraction(1,u)
```

$$\frac{\frac{1}{648}}{x+1} + \frac{\frac{1}{4} \, x + \frac{7}{16}}{(x+2)^2} + \frac{-\frac{17}{8} \, x^2 - 12 \, x - \frac{139}{8}}{(x+3)^3} +$$

$$\frac{\frac{607}{324} \, x^3 + \frac{10115}{432} \, x^2 + \frac{391}{4} \, x + \frac{44179}{324}}{(x + 4)^4}$$

Type: PartialFraction UnivariatePolynomial(x,Fraction Integer)

```
padicFraction %
```

$$\frac{\frac{1}{648}}{x+1} + \frac{\frac{1}{4}}{x+2} - \frac{\frac{1}{16}}{(x+2)^2} - \frac{\frac{17}{8}}{x+3} + \frac{\frac{3}{4}}{(x+3)^2} - \frac{\frac{1}{2}}{(x+3)^3} + \frac{\frac{607}{324}}{x+4} +$$

$$\frac{\frac{403}{432}}{(x + 4)^2} + \frac{\frac{13}{36}}{(x + 4)^3} + \frac{\frac{1}{12}}{(x + 4)^4}$$

Type: PartialFraction UnivariatePolynomial(x,Fraction Integer)

All see 9.29 on page 576 for examples of factor-free conversion of quotients to full partial fractions.

9.62   Permanent

The package `Permanent` provides the function **permanent** for square matrices. The **permanent** of a square matrix can be computed in the same way as the determinant by expansion of minors except that for the permanent the sign for each element is `1`, rather than being `1` if the row plus column indices is positive and `-1` otherwise. This function is much more difficult to compute efficiently than the **determinant**. An example of the use of **permanent** is the calculation of the n-th derangement number, defined to be the number of different possibilities for `n` couples to dance but never with their own spouse.

Consider an **n** by **n** matrix with entries **0** on the diagonal and **1** elsewhere. Think of the rows as one-half of each couple (for example, the males) and the columns the other half. The permanent of such a matrix gives the desired derangement number.

```
kn n ==
  r : MATRIX INT := new(n,n,1)
  for i in 1..n repeat
    r.i.i := 0
  r
```

```
                                                                Void
```

Here are some derangement numbers, which you see grow quite fast.

```
permanent(kn(5) ::  SQMATRIX(5,INT))
```

```
Compiling function kn with type PositiveInteger -> Matrix Integer
```

$$44$$

```
                                            Type:  PositiveInteger
```

```
[permanent(kn(n) ::  SQMATRIX(n,INT)) for n in 1..13]
```

```
Cannot compile conversion for types involving local variables.
   In particular, could not compile the expression involving
   :: SQMATRIX(n,INT)
AXIOM will attempt to step through and interpret the code.
```

$$[0, 1, 2, 9, 44, 265, 1854, 14833, 133496,$$
$$1334961, 14684570, 176214841, 2290792932]$$

```
                              Type:  List NonNegativeInteger
```

9.63   Polynomial

The domain constructor `Polynomial` (abbreviation: `POLY`) provides polynomials with an arbitrary number of unspecified variables.

It is used to create the default polynomial domains in Axiom. Here the coefficients are integers.

```
x + 1
```

$$x + 1$$

<div align="right">Type: Polynomial Integer</div>

Here the coefficients have type `Float`.

```
z - 2.3
```

$$z - 2.3$$

<div align="right">Type: Polynomial Float</div>

And here we have a polynomial in two variables with coefficients which have type `Fraction Integer`.

```
y**2 - z + 3/4
```

$$-z + y^2 + \frac{3}{4}$$

<div align="right">Type: Polynomial Fraction Integer</div>

The representation of objects of domains created by `Polynomial` is that of recursive univariate polynomials.[6]

This recursive structure is sometimes obvious from the display of a polynomial.

```
y **2 + x*y + y
```

$$y^2 + (x + 1) \, y$$

<div align="right">Type: Polynomial Integer</div>

In this example, you see that the polynomial is stored as a polynomial in `y` with coefficients that are polynomials in `x` with integer coefficients. In fact, you really don't need to worry about the representation unless you are working on an advanced application where it is critical. The polynomial types created from `DistributedMultivariatePolynomial` and `NewDistributedMultivariatePolynomial` (discussed in 9.16 on page 531) are stored and displayed in a non-recursive manner.

You see a "flat" display of the above polynomial by converting to one of those types.

---

[6]The term `univariate` means "one variable." `multivariate` means "possibly more than one variable."

```
% ::  DMP([y,x],INT)
```

$$y^2 + y\ x + y$$

<div align="center">Type: DistributedMultivariatePolynomial([y,x],Integer)</div>

We will demonstrate many of the polynomial facilities by using two polynomials with integer coefficients.

By default, the interpreter expands polynomial expressions, even if they are written in a factored format.

```
p := (y-1)**2 * x * z
```

$$\left(x\ y^2 - 2\ x\ y + x\right)\ z$$

<div align="right">Type: Polynomial Integer</div>

See 9.22 on page 546 to see how to create objects in factored form directly.

```
q := (y-1) * x * (z+5)
```

$$(x\ y - x)\ z + 5\ x\ y - 5\ x$$

<div align="right">Type: Polynomial Integer</div>

The fully factored form can be recovered by using **factor**.

```
factor(q)
```

$$x\ (y - 1)\ (z + 5)$$

<div align="center">Type: Factored Polynomial Integer</div>

This is the same name used for the operation to factor integers. Such reuse of names is called and makes it much easier to think of solving problems in general ways. Axiom facilities for factoring polynomials created with `Polynomial` are currently restricted to the integer and rational number coefficient cases. There are more complete facilities for factoring univariate polynomials: see **??** on page **??** in Section **??** on page **??**.

The standard arithmetic operations are available for polynomials.

```
p - q**2
```

$$\left(-x^2 \ y^2 + 2 \ x^2 \ y - x^2\right) \ z^2+$$

$$\left(\left(-10 \ x^2 + x\right) \ y^2 + \left(20 \ x^2 - 2 \ x\right) \ y - 10 \ x^2 + x\right) \ z-$$

$$25 \ x^2 \ y^2 + 50 \ x^2 \ y - 25 \ x^2$$

<div align="right">Type: Polynomial Integer</div>

The operation **gcd** is used to compute the greatest common divisor of two polynomials.

```
gcd(p,q)
```

$$x \ y - x$$

<div align="right">Type: Polynomial Integer</div>

In the case of `p` and `q`, the gcd is obvious from their definitions. We factor the gcd to show this relationship better.

```
factor %
```

$$x \ (y - 1)$$

<div align="right">Type: Factored Polynomial Integer</div>

The least common multiple is computed by using **lcm**.

```
lcm(p,q)
```

$$\left(x \ y^2 - 2 \ x \ y + x\right) \ z^2 + \left(5 \ x \ y^2 - 10 \ x \ y + 5 \ x\right) \ z$$

<div align="right">Type: Polynomial Integer</div>

Use **content** to compute the greatest common divisor of the coefficients of the polynomial.

```
content p
```

$$1$$

<div align="right">Type: PositiveInteger</div>

Many of the operations on polynomials require you to specify a variable. For example, **resultant** requires you to give the variable in which the polynomials should be expressed.

This computes the resultant of the values of `p` and `q`, considering them as polynomials in the variable `z`. They do not share a root when thought of as polynomials in `z`.

```
resultant(p,q,z)
```

$$5\ x^2\ y^3 - 15\ x^2\ y^2 + 15\ x^2\ y - 5\ x^2$$

<div align="right">Type: Polynomial Integer</div>

This value is `0` because as polynomials in `x` the polynomials have a common root.

```
resultant(p,q,x)
```

$$0$$

<div align="right">Type: Polynomial Integer</div>

The data type used for the variables created by `Polynomial` is `Symbol`. As mentioned above, the representation used by `Polynomial` is recursive and so there is a main variable for nonconstant polynomials.

The operation **mainVariable** returns this variable. The return type is actually a union of `Symbol` and `"failed"`.

```
mainVariable p
```

$$z$$

<div align="right">Type: Union(Symbol,...)</div>

The latter branch of the union is be used if the polynomial has no variables, that is, is a constant.

```
mainVariable(1 ::  POLY INT)
```

```
"failed"
```

<div align="right">Type: Union("failed",...)</div>

You can also use the predicate **ground?** to test whether a polynomial is in fact a member of its ground ring.

```
ground?  p
```

$$false$$

                                              Type: Boolean

```
ground?(1 ::  POLY INT)
```

$$true$$

                                              Type: Boolean

The complete list of variables actually used in a particular polynomial is returned by **variables**. For constant polynomials, this list is empty.

```
variables p
```

$$[z, y, x]$$

                                              Type: List Symbol

The **degree** operation returns the degree of a polynomial in a specific variable.

```
degree(p,x)
```

$$1$$

                                              Type: PositiveInteger

```
degree(p,y)
```

$$2$$

                                              Type: PositiveInteger

```
degree(p,z)
```

$$1$$

If you give a list of variables for the second argument, a list of the degrees in those variables is returned.

```
degree(p,[x,y,z])
```

$$[1, 2, 1]$$

Type: List NonNegativeInteger

The minimum degree of a variable in a polynomial is computed using **minimumDegree**.

```
minimumDegree(p,z)
```

$$1$$

Type: PositiveInteger

The total degree of a polynomial is returned by **totalDegree**.

```
totalDegree p
```

$$4$$

Type: PositiveInteger

It is often convenient to think of a polynomial as a leading monomial plus the remaining terms.

```
leadingMonomial p
```

$$x \, y^2 \, z$$

Type: Polynomial Integer

The **reductum** operation returns a polynomial consisting of the sum of the monomials after the first.

```
reductum p
```

$$(-2 \, x \, y + x) \, z$$

These have the obvious relationship that the original polynomial is equal to the leading monomial plus the reductum.

```
p - leadingMonomial p - reductum p
```

$$0$$

Type: Polynomial Integer

The value returned by **leadingMonomial** includes the coefficient of that term. This is extracted by using **leadingCoefficient** on the original polynomial.

```
leadingCoefficient p
```

$$1$$

Type: PositiveInteger

The operation **eval** is used to substitute a value for a variable in a polynomial.

```
p
```

$$\left(x\, y^2 - 2\, x\, y + x\right)\, z$$

Type: Polynomial Integer

This value may be another variable, a constant or a polynomial.

```
eval(p,x,w)
```

$$\left(w\, y^2 - 2\, w\, y + w\right)\, z$$

Type: Polynomial Integer

```
eval(p,x,1)
```

$$\left(y^2 - 2\, y + 1\right)\, z$$

Type: Polynomial Integer

Actually, all the things being substituted are just polynomials, some more trivial than others.

```
eval(p,x,y**2 - 1)
```

$$\left(y^4 - 2\ y^3 + 2\ y - 1\right)\ z$$

<div align="right">Type: Polynomial Integer</div>

Derivatives are computed using the **D** operation.

```
D(p,x)
```

$$\left(y^2 - 2\ y + 1\right)\ z$$

<div align="right">Type: Polynomial Integer</div>

The first argument is the polynomial and the second is the variable.

```
D(p,y)
```

$$\left(2\ x\ y - 2\ x\right)\ z$$

<div align="right">Type: Polynomial Integer</div>

Even if the polynomial has only one variable, you must specify it.

```
D(p,z)
```

$$x\ y^2 - 2\ x\ y + x$$

<div align="right">Type: Polynomial Integer</div>

Integration of polynomials is similar and the **integrate** operation is used.

Integration requires that the coefficients support division. Consequently, Axiom converts polynomials over the integers to polynomials over the rational numbers before integrating them.

```
integrate(p,y)
```

$$\left(\frac{1}{3}\ x\ y^3 - x\ y^2 + x\ y\right)\ z$$

It is not possible, in general, to divide two polynomials. In our example using polynomials over the integers, the operation **monicDivide** divides a polynomial by a monic polynomial (that is, a polynomial with leading coefficient equal to 1). The result is a record of the quotient and remainder of the division.

You must specify the variable in which to express the polynomial.

```
qr := monicDivide(p,x+1,x)
```

$$\left[quotient = \left(y^2 - 2\,y + 1\right)\,z, remainder = \left(-y^2 + 2\,y - 1\right)\,z\right]$$

Type: Record(quotient: Polynomial Integer,remainder:
Polynomial Integer)

The selectors of the components of the record are `quotient` and `remainder`. Issue this to extract the remainder.

```
qr.remainder
```

$$\left(-y^2 + 2\,y - 1\right)\,z$$

Type: Polynomial Integer

Now that we can extract the components, we can demonstrate the relationship among them and the arguments to our original expression `qr := monicDivide(p,x+1,x)`.

```
p - ((x+1) * qr.quotient + qr.remainder)
```

$$0$$

Type: Polynomial Integer

If the "/" operator is used with polynomials, a fraction object is created. In this example, the result is an object of type `Fraction Polynomial Integer`.

```
p/q
```

$$\frac{(y - 1)\,z}{z + 5}$$

Type: Fraction Polynomial Integer

If you use rational numbers as polynomial coefficients, the resulting object is of type `Polynomial Fraction Integer`.

```
(2/3) * x**2 - y + 4/5
```

$$-y + \frac{2}{3}\ x^2 + \frac{4}{5}$$

<div align="right">Type: Polynomial Fraction Integer</div>

This can be converted to a fraction of polynomials and back again, if required.

```
% ::  FRAC POLY INT
```

$$\frac{-15\ y + 10\ x^2 + 12}{15}$$

<div align="right">Type: Fraction Polynomial Integer</div>

```
% ::  POLY FRAC INT
```

$$-y + \frac{2}{3}\ x^2 + \frac{4}{5}$$

<div align="right">Type: Polynomial Fraction Integer</div>

To convert the coefficients to floating point, map the `numeric` operation on the coefficients of the polynomial.

```
map(numeric,%)
```

$$-1.0\ y + 0.66666666666666666667\ x^2 + 0.8$$

<div align="right">Type: Polynomial Float</div>

For more information on related topics, see . You can also issue the system command `)show Polynomial` to display the full list of operations defined by `Polynomial`.

9.64 Quaternion

The domain constructor `Quaternion` implements quaternions over commutative rings. For information on related topics, see 9.11 on page 501 and 9.56 on page 712. You can also issue the system command `)show Quaternion` to display the full list of operations defined by `Quaternion`.

The basic operation for creating quaternions is **quatern**. This is a quaternion over the rational numbers.

```
q := quatern(2/11,-8,3/4,1)
```

$$\frac{2}{11} - 8\ i + \frac{3}{4}\ j + k$$

<div align="right">Type: Quaternion Fraction Integer</div>

The four arguments are the real part, the `i` imaginary part, the `j` imaginary part, and the `k` imaginary part, respectively.

```
[real q, imagI q, imagJ q, imagK q]
```

$$\left[\frac{2}{11}, -8, \frac{3}{4}, 1\right]$$

<div align="right">Type: List Fraction Integer</div>

Because `q` is over the rationals (and nonzero), you can invert it.

```
inv q
```

$$\frac{352}{126993} + \frac{15488}{126993}\ i - \frac{484}{42331}\ j - \frac{1936}{126993}\ k$$

<div align="right">Type: Quaternion Fraction Integer</div>

The usual arithmetic (ring) operations are available

```
q**6
```

$$-\frac{2029490709319345}{7256313856} - \frac{48251690851}{1288408}\ i + \frac{144755072553}{41229056}\ j + \frac{48251690851}{10307264}\ k$$

<div align="right">Type: Quaternion Fraction Integer</div>

```
r := quatern(-2,3,23/9,-89); q + r
```

$$-\frac{20}{11} - 5\ i + \frac{119}{36}\ j - 88\ k$$

<div align="right">Type: Quaternion Fraction Integer</div>

In general, multiplication is not commutative.

```
q * r - r * q
```

$$-\frac{2495}{18}\ i - 1418\ j - \frac{817}{18}\ k$$

<div align="right">Type: Quaternion Fraction Integer</div>

There are no predefined constants for the imaginary i, j, and k parts, but you can easily define them.

```
i:=quatern(0,1,0,0); j:=quatern(0,0,1,0); k:=quatern(0,0,0,1)
```

$$k$$

<div align="right">Type: Quaternion Integer</div>

These satisfy the normal identities.

```
[i*i, j*j, k*k, i*j, j*k, k*i, q*i]
```

$$\left[-1, -1, -1, k, i, j, 8 + \frac{2}{11}\ i + j - \frac{3}{4}\ k\right]$$

<div align="right">Type: List Quaternion Fraction Integer</div>

The norm is the quaternion times its conjugate.

```
norm q
```

$$\frac{126993}{1936}$$

<div align="right">Type: Fraction Integer</div>

```
conjugate q
```

$$\frac{2}{11} + 8\,i - \frac{3}{4}\,j - k$$

<div align="center">Type: Quaternion Fraction Integer</div>

```
q * %
```

$$\frac{126993}{1936}$$

<div align="center">Type: Quaternion Fraction Integer</div>

9.65 RadixExpansion

It possible to expand numbers in general bases.

Here we expand 111 in base 5. This means

$$10^2 + 10^1 + 10^0 = 4 \cdot 5^2 + 2 \cdot 5^1 + 5^0$$

```
111::RadixExpansion(5)
```

<div align="center">421</div>

<div align="center">Type: RadixExpansion 5</div>

You can expand fractions to form repeating expansions.

```
(5/24)::RadixExpansion(2)
```

$$0.001\overline{10}$$

<div align="center">Type: RadixExpansion 2</div>

```
(5/24)::RadixExpansion(3)
```

$$0.0\overline{12}$$

<div align="center">Type: RadixExpansion 3</div>

`(5/24)::RadixExpansion(8)`

$$0.1\overline{52}$$

Type: RadixExpansion 8

`(5/24)::RadixExpansion(10)`

$$0.208\overline{3}$$

Type: RadixExpansion 10

For bases from 11 to 36 the letters A through Z are used.

`(5/24)::RadixExpansion(12)`

$$0.26$$

Type: RadixExpansion 12

`(5/24)::RadixExpansion(16)`

$$0.3\overline{5}$$

Type: RadixExpansion 16

`(5/24)::RadixExpansion(36)`

$$0.7I$$

Type: RadixExpansion 36

For bases greater than 36, the ragits are separated by blanks.

`(5/24)::RadixExpansion(38)`

$$0 . 7 \ 34 \ 31 \ \overline{25 \ 12}$$

Type: RadixExpansion 38

The `RadixExpansion` type provides operations to obtain the individual ragits. Here is a rational number in base 8.

```
a := (76543/210)::RadixExpansion(8)
```

$$554.3\overline{7307}$$

<div style="text-align: right">Type: RadixExpansion 8</div>

The operation **wholeRagits** returns a list of the ragits for the integral part of the number.

```
w := wholeRagits a
```

$$[5, 5, 4]$$

<div style="text-align: right">Type: List Integer</div>

The operations **prefixRagits** and **cycleRagits** return lists of the initial and repeating ragits in the fractional part of the number.

```
f0 := prefixRagits a
```

$$[3]$$

<div style="text-align: right">Type: List Integer</div>

```
f1 := cycleRagits a
```

$$[7, 3, 0, 7]$$

<div style="text-align: right">Type: List Integer</div>

You can construct any radix expansion by giving the whole, prefix and cycle parts. The declaration is necessary to let Axiom know the base of the ragits.

```
u:RadixExpansion(8):=wholeRadix(w)+fractRadix(f0,f1)
```

$$554.3\overline{7307}$$

<div style="text-align: right">Type: RadixExpansion 8</div>

If there is no repeating part, then the list `[0]` should be used.

```
v:  RadixExpansion(12) := fractRadix([1,2,3,11], [0])
```

$$0.123B\overline{0}$$

<div align="right">Type: RadixExpansion 12</div>

If you are not interested in the repeating nature of the expansion, an infinite stream of ragits can be obtained using **fractRagits**.

```
fractRagits(u)
```

$$\left[3, 7, \overline{3, 0, 7, 7}\right]$$

<div align="right">Type: Stream Integer</div>

Of course, it's possible to recover the fraction representation:

```
a ::  Fraction(Integer)
```

$$\frac{76543}{210}$$

<div align="right">Type: Fraction Integer</div>

More examples of expansions are available in 9.15 on page 529, 9.4 on page 468, and 9.33 on page 586.

9.66   RealClosure

The Real Closure 1.0 package provided by Renaud Rioboo (Renaud.Rioboo@lip6.fr) consists of different packages, categories and domains :

> The package `RealPolynomialUtilitiesPackage` which needs a `Field` F and a `UnivariatePolynomialCategory` domain with coefficients in F. It computes some simple functions such as Sturm and Sylvester sequences (**sturmSequence**, **sylvesterSequence**).

> The category `RealRootCharacterizationCategory` provides abstract functions to work with "real roots" of univariate polynomials. These resemble variables with some functionality needed to compute important operations.

The category `RealClosedField` provides common operations available over real closed fiels. These include finding all the roots of a univariate polynomial, taking square (and higher) roots, ...

The domain `RightOpenIntervalRootCharacterization` is the main code that provides the functionality of `RealRootCharacterizationCategory` for the case of archimedean fields. Abstract roots are encoded with a left closed right open interval containing the root together with a defining polynomial for the root.

The `RealClosure` domain is the end-user code. It provides usual arithmetic with real algebraic numbers, along with the functionality of a real closed field. It also provides functions to approximate a real algebraic number by an element of the base field. This approximation may either be absolute (**approximate**) or relative (**relativeApprox**).

CAVEATS

Since real algebraic expressions are stored as depending on "real roots" which are managed like variables, there is an ordering on these. This ordering is dynamical in the sense that any new algebraic takes precedence over older ones. In particular every creation function raises a new "real root". This has the effect that when you type something like `sqrt(2) + sqrt(2)` you have two new variables which happen to be equal. To avoid this name the expression such as in `s2 := sqrt(2) ; s2 + s2`

Also note that computing times depend strongly on the ordering you implicitly provide. Please provide algebraics in the order which seems most natural to you.

LIMITATIONS

This packages uses algorithms which are published in [1] and [2] which are based on field arithmetics, in particular for polynomial gcd related algorithms. This can be quite slow for high degree polynomials and subresultants methods usually work best. Beta versions of the package try to use these techniques in a better way and work significantly faster. These are mostly based on unpublished algorithms and cannot be distributed. Please contact the author if you have a particular problem to solve or want to use these versions.

Be aware that approximations behave as post-processing and that all computations are done exactly. They can thus be quite time consuming when depending on several "real roots".

REFERENCES

[1] R. Rioboo : Real Algebraic Closure of an ordered Field : Implementation in Axiom. In proceedings of the ISSAC'92 Conference, Berkeley 1992 pp. 206-215.

[2] Z. Ligatsikas, R. Rioboo, M. F. Roy : Generic computation of the real closure of an ordered field. In Mathematics and Computers in Simulation Volume 42, Issue 4-6, November 1996.

EXAMPLES

We shall work with the real closure of the ordered field of rational numbers.

```
Ran := RECLOS(FRAC INT)
```

<div align="center">RealClosure Fraction Integer</div>

<div align="right">Type: Domain</div>

Some simple signs for square roots, these correspond to an extension of degree 16 of the rational numbers. Examples provided by J. Abbot.

```
fourSquares(a:Ran,b:Ran,c:Ran,d:Ran):Ran == sqrt(a)+sqrt(b) -
sqrt(c)-sqrt(d)
```

```
Function declaration fourSquares : (RealClosure Fraction Integer,
   RealClosure Fraction Integer,RealClosure Fraction Integer,
   RealClosure Fraction Integer) -> RealClosure Fraction Integer has
   been added to workspace.
```

<div align="right">Void</div>

These produce values very close to zero.

```
squareDiff1 := fourSquares(73,548,60,586)
```

$$-\sqrt{586} - \sqrt{60} + \sqrt{548} + \sqrt{73}$$

<div align="right">Type: RealClosure Fraction Integer</div>

```
recip(squareDiff1)
```

$$\left(\left(54602\ \sqrt{548} + 149602\ \sqrt{73}\right)\ \sqrt{60} + 49502\ \sqrt{73}\ \sqrt{548} + 9900895\right)\ \sqrt{586} +$$

$$\left(154702\ \sqrt{73}\ \sqrt{548} + 30941947\right)\ \sqrt{60} + 10238421\ \sqrt{548} + 28051871\ \sqrt{73}$$

<div align="right">Type: Union(RealClosure Fraction Integer,...)</div>

```
sign(squareDiff1)
```

<div align="center">1</div>

squareDiff2 := fourSquares(165,778,86,990)

$$-\sqrt{990} - \sqrt{86} + \sqrt{778} + \sqrt{165}$$

Type: RealClosure Fraction Integer

recip(squareDiff2)

$$\left(\left(556778 \ \sqrt{778} + 1209010 \ \sqrt{165}\right) \ \sqrt{86} + \right.$$

$$\left.401966 \ \sqrt{165} \ \sqrt{778} + 144019431\right) \ \sqrt{990} +$$

$$\left(1363822 \ \sqrt{165} \ \sqrt{778} + 488640503\right) \ \sqrt{86} +$$

$$162460913 \ \sqrt{778} + 352774119 \ \sqrt{165}$$

Type: Union(RealClosure Fraction Integer,...)

sign(squareDiff2)

$$1$$

Type: PositiveInteger

squareDiff3 := fourSquares(217,708,226,692)

$$-\sqrt{692} - \sqrt{226} + \sqrt{708} + \sqrt{217}$$

Type: RealClosure Fraction Integer

recip(squareDiff3)

$$\left(\left(-34102 \ \sqrt{708} - 61598 \ \sqrt{217}\right) \ \sqrt{226} - \right.$$

$$\left.34802 \ \sqrt{217} \ \sqrt{708} - 13641141\right) \ \sqrt{692} +$$

$$\left(-60898 \ \sqrt{217} \ \sqrt{708} - 23869841\right) \ \sqrt{226} - $$

$$13486123 \ \sqrt{708} - 24359809 \ \sqrt{217}$$

$$\text{Type: Union(RealClosure Fraction Integer,...)}$$

sign(squareDiff3)

$$-1$$

$$\text{Type: Integer}$$

squareDiff4 := fourSquares(155,836,162,820)

$$-\sqrt{820} - \sqrt{162} + \sqrt{836} + \sqrt{155}$$

$$\text{Type: RealClosure Fraction Integer}$$

recip(squareDiff4)

$$\left(\left(-37078 \ \sqrt{836} - 86110 \ \sqrt{155} \right) \ \sqrt{162} - \right.$$

$$37906 \ \sqrt{155} \ \sqrt{836} - 13645107 \Big) \ \sqrt{820} +$$

$$\left(-85282 \ \sqrt{155} \ \sqrt{836} - 30699151 \right) \ \sqrt{162} -$$

$$13513901 \ \sqrt{836} - 31384703 \ \sqrt{155}$$

$$\text{Type: Union(RealClosure Fraction Integer,...)}$$

sign(squareDiff4)

$$-1$$

$$\text{Type: Integer}$$

squareDiff5 := fourSquares(591,772,552,818)

$$-\sqrt{818} - \sqrt{552} + \sqrt{772} + \sqrt{591}$$

$$\text{Type: RealClosure Fraction Integer}$$

```
recip(squareDiff5)
```

$$\Big(\big(70922 \ \sqrt{772} + 81058 \ \sqrt{591}\big) \ \sqrt{552} +$$

$$68542 \ \sqrt{591} \ \sqrt{772} + 46297673 \Big) \ \sqrt{818} +$$

$$\Big(83438 \ \sqrt{591} \ \sqrt{772} + 56359389 \Big) \ \sqrt{552} +$$

$$47657051 \ \sqrt{772} + 54468081 \ \sqrt{591}$$

<div align="center">Type: Union(RealClosure Fraction Integer,...)</div>

```
sign(squareDiff5)
```

$$1$$

<div align="center">Type: PositiveInteger</div>

```
squareDiff6 := fourSquares(434,1053,412,1088)
```

$$-\sqrt{1088} - \sqrt{412} + \sqrt{1053} + \sqrt{434}$$

<div align="center">Type: RealClosure Fraction Integer</div>

```
recip(squareDiff6)
```

$$\Big(\big(115442 \ \sqrt{1053} + 179818 \ \sqrt{434}\big) \ \sqrt{412} +$$

$$112478 \ \sqrt{434} \ \sqrt{1053} + 76037291 \Big) \ \sqrt{1088} +$$

$$\Big(182782 \ \sqrt{434} \ \sqrt{1053} + 123564147 \Big) \ \sqrt{412} +$$

$$77290639 \ \sqrt{1053} + 120391609 \ \sqrt{434}$$

<div align="center">Type: Union(RealClosure Fraction Integer,...)</div>

```
sign(squareDiff6)
```

$$1$$

squareDiff7 := fourSquares(514,1049,446,1152)

$$-\sqrt{1152} - \sqrt{446} + \sqrt{1049} + \sqrt{514}$$

Type: RealClosure Fraction Integer

recip(squareDiff7)

$$\left(\left(349522 \ \sqrt{1049} + 499322 \ \sqrt{514}\right) \ \sqrt{446} + \right.$$

$$\left. 325582 \ \sqrt{514} \ \sqrt{1049} + 239072537\right) \ \sqrt{1152} +$$

$$\left(523262 \ \sqrt{514} \ \sqrt{1049} + 384227549\right) \ \sqrt{446} +$$

$$250534873 \ \sqrt{1049} + 357910443 \ \sqrt{514}$$

Type: Union(RealClosure Fraction Integer,...)

sign(squareDiff7)

$$1$$

Type: PositiveInteger

squareDiff8 := fourSquares(190,1751,208,1698)

$$-\sqrt{1698} - \sqrt{208} + \sqrt{1751} + \sqrt{190}$$

Type: RealClosure Fraction Integer

recip(squareDiff8)

$$\left(\left(-214702 \ \sqrt{1751} - 651782 \ \sqrt{190}\right) \ \sqrt{208} - \right.$$

$$\left. 224642 \ \sqrt{190} \ \sqrt{1751} - 129571901\right) \ \sqrt{1698} +$$

$$\left(-641842 \ \sqrt{190} \ \sqrt{1751} - 370209881\right) \ \sqrt{208} -$$

$$127595865 \ \sqrt{1751} - 387349387 \ \sqrt{190}$$

$$\text{Type: Union(RealClosure Fraction Integer,...)}$$

```
sign(squareDiff8)
```

$$-1$$

$$\text{Type: Integer}$$

This should give three digits of precision

```
relativeApprox(squareDiff8,10**(-3))::Float
```

$$-0.2340527771\ 5937700123E-10$$

$$\text{Type: Float}$$

The sum of these 4 roots is 0

```
l := allRootsOf((x**2-2)**2-2)$Ran
```

$$[\%A33, \%A34, \%A35, \%A36]$$

$$\text{Type: List RealClosure Fraction Integer}$$

Check that they are all roots of the same polynomial

```
removeDuplicates map(mainDefiningPolynomial,l)
```

$$\left[?^4 - 4\ ?^2 + 2\right]$$

Type: List Union(SparseUnivariatePolynomial RealClosure Fraction
Integer,"failed")

We can see at a glance that they are separate roots

```
map(mainCharacterization,l)
```

$$[[-2, -1[, [-1, 0[, [0, 1[, [1, 2[]$$

Type: List Union(RightOpenIntervalRootCharacterization(
RealClosure Fraction Integer, SparseUnivariatePolynomial
RealClosure Fraction Integer), "failed")

Check the sum and product

```
[reduce(+,l),reduce(*,l)-2]
```

$$[0, 0]$$

<div align="center">Type: List RealClosure Fraction Integer</div>

A more complicated test that involve an extension of degree 256. This is a way of checking nested radical identities.

```
(s2, s5, s10) := (sqrt(2)$Ran, sqrt(5)$Ran, sqrt(10)$Ran)
```

$$\sqrt{10}$$

<div align="center">Type: RealClosure Fraction Integer</div>

```
eq1:=sqrt(s10+3)*sqrt(s5+2) - sqrt(s10-3)*sqrt(s5-2) =
sqrt(10*s2+10)
```

$$-\sqrt{\sqrt{10}-3}\,\sqrt{\sqrt{5}-2}+\sqrt{\sqrt{10}+3}\,\sqrt{\sqrt{5}+2}=\sqrt{10\,\sqrt{2}+10}$$

<div align="center">Type: Equation RealClosure Fraction Integer</div>

```
eq1::Boolean
```

<div align="center">true</div>

<div align="right">Type: Boolean</div>

```
eq2:=sqrt(s5+2)*sqrt(s2+1) - sqrt(s5-2)*sqrt(s2-1) =
sqrt(2*s10+2)
```

$$-\sqrt{\sqrt{5}-2}\,\sqrt{\sqrt{2}-1}+\sqrt{\sqrt{5}+2}\,\sqrt{\sqrt{2}+1}=\sqrt{2\,\sqrt{10}+2}$$

<div align="center">Type: Equation RealClosure Fraction Integer</div>

```
eq2::Boolean
```

$$true$$

Type: Boolean

Some more examples from J. M. Arnaudies

```
s3 := sqrt(3)$Ran
```

$$\sqrt{3}$$

Type: RealClosure Fraction Integer

```
s7:= sqrt(7)$Ran
```

$$\sqrt{7}$$

Type: RealClosure Fraction Integer

```
e1 := sqrt(2*s7-3*s3,3)
```

$$\sqrt[3]{2\ \sqrt{7} - 3\ \sqrt{3}}$$

Type: RealClosure Fraction Integer

```
e2 := sqrt(2*s7+3*s3,3)
```

$$\sqrt[3]{2\ \sqrt{7} + 3\ \sqrt{3}}$$

Type: RealClosure Fraction Integer

This should be null

```
e2-e1-s3
```

$$0$$

Type: RealClosure Fraction Integer

A quartic polynomial

```
pol :  UP(x,Ran) := x**4+(7/3)*x**2+30*x-(100/3)
```

$$x^4 + \frac{7}{3}\, x^2 + 30\, x - \frac{100}{3}$$

    Type: UnivariatePolynomial(x,RealClosure Fraction Integer)

Add some cubic roots

```
r1 := sqrt(7633)$Ran
```

$$\sqrt{7633}$$

                Type: RealClosure Fraction Integer

```
alpha := sqrt(5*r1-436,3)/3
```

$$\frac{1}{3}\ \sqrt[3]{5\ \sqrt{7633} - 436}$$

                Type: RealClosure Fraction Integer

```
beta := -sqrt(5*r1+436,3)/3
```

$$-\frac{1}{3}\ \sqrt[3]{5\ \sqrt{7633} + 436}$$

                Type: RealClosure Fraction Integer

this should be null

```
pol.(alpha+beta-1/3)
```

$$0$$

                Type: RealClosure Fraction Integer

A quintic polynomial

```
qol :  UP(x,Ran) := x**5+10*x**3+20*x+22
```

$$x^5 + 10\, x^3 + 20\, x + 22$$

```
      Type:  UnivariatePolynomial(x,RealClosure Fraction Integer)
```

Add some cubic roots

```
r2 := sqrt(153)$Ran
```

$$\sqrt{153}$$

<div align="center">Type: RealClosure Fraction Integer</div>

```
alpha2 := sqrt(r2-11,5)
```

$$\sqrt[5]{\sqrt{153} - 11}$$

<div align="center">Type: RealClosure Fraction Integer</div>

```
beta2 := -sqrt(r2+11,5)
```

$$-\sqrt[5]{\sqrt{153} + 11}$$

<div align="center">Type: RealClosure Fraction Integer</div>

this should be null

```
qol(alpha2+beta2)
```

$$0$$

<div align="center">Type: RealClosure Fraction Integer</div>

Finally, some examples from the book Computer Algebra by Davenport, Siret and Tournier (page 77). The last one is due to Ramanujan.

```
dst1:=sqrt(9+4*s2)=1+2*s2
```

$$\sqrt{4 \sqrt{2} + 9} = 2 \sqrt{2} + 1$$

<div align="center">Type: Equation RealClosure Fraction Integer</div>

dst1::Boolean

$$true$$

                                                Type: Boolean

s6:Ran:=sqrt 6

$$\sqrt{6}$$

                                    Type: RealClosure Fraction Integer

dst2:=sqrt(5+2*s6)+sqrt(5-2*s6) = 2*s3

$$\sqrt{-2\ \sqrt{6}+5}+\sqrt{2\ \sqrt{6}+5}=2\ \sqrt{3}$$

                            Type: Equation RealClosure Fraction Integer

dst2::Boolean

$$true$$

                                                Type: Boolean

s29:Ran:=sqrt 29

$$\sqrt{29}$$

                                    Type: RealClosure Fraction Integer

dst4:=sqrt(16-2*s29+2*sqrt(55-10*s29)) =
sqrt(22+2*s5)-sqrt(11+2*s29)+s5

$$\sqrt{2\ \sqrt{-10\ \sqrt{29}+55}-2\ \sqrt{29}+16}=-\sqrt{2\ \sqrt{29}+11}+\sqrt{2\ \sqrt{5}+22}+\sqrt{5}$$

                            Type: Equation RealClosure Fraction Integer

dst4::Boolean

$$\text{true}$$

<div align="right">Type: Boolean</div>

```
dst6:=sqrt((112+70*s2)+(46+34*s2)*s5) = (5+4*s2)+(3+s2)*s5
```

$$\sqrt{\left(34\ \sqrt{2}+46\right)\ \sqrt{5}+70\ \sqrt{2}+112}=\left(\sqrt{2}+3\right)\ \sqrt{5}+4\ \sqrt{2}+5$$

<div align="right">Type: Equation RealClosure Fraction Integer</div>

```
dst6::Boolean
```

$$\text{true}$$

<div align="right">Type: Boolean</div>

```
f3:Ran:=sqrt(3,5)
```

$$\sqrt[5]{3}$$

<div align="right">Type: RealClosure Fraction Integer</div>

```
f25:Ran:=sqrt(1/25,5)
```

$$\sqrt[5]{\frac{1}{25}}$$

<div align="right">Type: RealClosure Fraction Integer</div>

```
f32:Ran:=sqrt(32/5,5)
```

$$\sqrt[5]{\frac{32}{5}}$$

<div align="right">Type: RealClosure Fraction Integer</div>

```
f27:Ran:=sqrt(27/5,5)
```

$$\sqrt[5]{\dfrac{27}{5}}$$

<div align="right">Type: RealClosure Fraction Integer</div>

```
dst5:=sqrt((f32-f27,3)) = f25*(1+f3-f3**2)
```

$$\sqrt[3]{-\sqrt[5]{\dfrac{27}{5}}+\sqrt[5]{\dfrac{32}{5}}} = \left(-\sqrt[5]{3}^{2}+\sqrt[5]{3}+1\right)\ \sqrt[5]{\dfrac{1}{25}}$$

<div align="right">Type: Equation RealClosure Fraction Integer</div>

```
dst5::Boolean
```

<div align="center">true</div>

<div align="right">Type: Boolean</div>

9.67   RegularTriangularSet

The `RegularTriangularSet` domain constructor implements regular triangular sets. These particular triangular sets were introduced by M. Kalkbrener (1991) in his PhD Thesis under the name regular chains. Regular chains and their related concepts are presented in the paper "On the Theories of Triangular sets" By P. Aubry, D. Lazard and M. Moreno Maza (to appear in the Journal of Symbolic Computation). The `RegularTriangularSet` constructor also provides a new method (by the third author) for solving polynomial system by means of regular chains. This method has two ways of solving. One has the same specifications as Kalkbrener's algorithm (1991) and the other is closer to Lazard's method (Discr. App. Math, 1991). Moreover, this new method removes redundant component from the decompositions when this is not *too expensive*. This is always the case with squarefree regular chains. So if you want to obtain decompositions without redundant components just use the `SquareFreeRegularTriangularSet` domain constructor or the `LazardSetSolvingPackage` package constructor. See also the `LexTriangularPackage` and `ZeroDimensionalSolvePackage` for the case of algebraic systems with a finite number of (complex) solutions.

One of the main features of regular triangular sets is that they naturally define towers of simple extensions of a field. This allows to perform with multivariate polynomials the same kind of operations as one can do in an `EuclideanDomain`.

The `RegularTriangularSet` constructor takes four arguments. The first one, **R**, is the coefficient ring of the polynomials; it must belong to the category `GcdDomain`. The second one, **E**, is the exponent monoid of the polynomials; it must belong to the category `OrderedAbelianMonoidSup`. the third one, **V**, is the ordered set of variables; it must belong to the category `OrderedSet`. The last one is the polynomial ring; it must belong to the category `RecursivePolynomialCategory(R,E,V)`. The abbreviation for `RegularTriangularSet` is REGSET. See also the constructor `RegularChain` which only takes two arguments, the coefficient ring and the ordered set of variables; in that case, polynomials are necessarily built with the `NewSparseMultivariatePolynomial` domain constructor.

We shall explain now how to use the constructor REGSET and how to read the decomposition of a polynomial system by means of regular sets.

Let us give some examples. We start with an easy one (Donati-Traverso) in order to understand the two ways of solving polynomial systems provided by the REGSET constructor.

Define the coefficient ring.

```
R := Integer
```

$$Integer$$

<div align="right">Type: Domain</div>

Define the list of variables,

```
ls :  List Symbol := [x,y,z,t]
```

$$[x, y, z, t]$$

<div align="right">Type: List Symbol</div>

and make it an ordered set;

```
V := OVAR(ls)
```

$$OrderedVariableList \ [x,y,z,t]$$

<div align="right">Type: Domain</div>

then define the exponent monoid.

```
E := IndexedExponents V
```

IndexedExponents OrderedVariableList [x,y,z,t]

                                                      Type: Domain

Define the polynomial ring.

P := NSMP(R, V)

   NewSparseMultivariatePolynomial(Integer,OrderedVariableList [x,y,z,t])

                                                      Type: Domain

Let the variables be polynomial.

x: P := 'x

$$x$$

                Type: NewSparseMultivariatePolynomial(Integer,
                            OrderedVariableList [x,y,z,t])

y: P := 'y

$$y$$

                Type: NewSparseMultivariatePolynomial(Integer,
                            OrderedVariableList [x,y,z,t])

z: P := 'z

$$z$$

                Type: NewSparseMultivariatePolynomial(Integer,
                            OrderedVariableList [x,y,z,t])

t: P := 't

$$t$$

                Type: NewSparseMultivariatePolynomial(Integer,
                            OrderedVariableList [x,y,z,t])

Now call the `RegularTriangularSet` domain constructor.

```
T := REGSET(R,E,V,P)
```

$$\begin{aligned} &\text{RegularTriangularSet(Integer,} \\ &\quad \text{IndexedExponentsOrderedVariableList}[x, y, z, t], \\ &\quad \text{OrderedVariableList}[x, y, z, t], \\ &\quad \text{NewSparseMultivariatePolynomial(Integer,} \\ &\quad \text{OrderedVariableList}[x, y, z, t])) \end{aligned}$$

                                            Type: Domain

Define a polynomial system.

```
p1 := x ** 31 - x ** 6 - x - y
```

$$x^{31} - x^6 - x - y$$

                                            Type:
        NewSparseMultivariatePolynomial(Integer,OrderedVariableList
                                            [x,y,z,t])

```
p2 := x ** 8 - z
```

$$x^8 - z$$

                                            Type:
        NewSparseMultivariatePolynomial(Integer,OrderedVariableList
                                            [x,y,z,t])

```
p3 := x ** 10 - t
```

$$x^{10} - t$$

            Type: NewSparseMultivariatePolynomial(Integer,
                            OrderedVariableList [x,y,z,t])

```
lp := [p1, p2, p3]
```

$$\left[x^{31} - x^6 - x - y, x^8 - z, x^{10} - t\right]$$

```
            Type:  List NewSparseMultivariatePolynomial( Integer,
                                    OrderedVariableList [x,y,z,t])
```

First of all, let us solve this system in the sense of Kalkbrener.

```
zeroSetSplit(lp)$T
```

$$\left[\left\{z^5 - t^4, t\ z\ y^2 + 2\ z^3\ y - t^8 + 2\ t^5 + t^3 - t^2, \left(t^4 - t\right)\ x - t\ y - z^2\right\}\right]$$

```
    Type:  List RegularTriangularSet( Integer, IndexedExponents
  OrderedVariableList [x,y,z,t], OrderedVariableList [x,y,z,t],
  NewSparseMultivariatePolynomial( Integer, OrderedVariableList
                                                   [x,y,z,t]))
```

And now in the sense of Lazard (or Wu and other authors).

```
lts := zeroSetSplit(lp,false)$T
```

$$\left[\left\{z^5 - t^4, t\ z\ y^2 + 2\ z^3\ y - t^8 + 2\ t^5 + t^3 - t^2, \left(t^4 - t\right)\ x - t\ y - z^2\right\},\right.$$

$$\left.\left\{t^3 - 1, z^5 - t, t\ z\ y^2 + 2\ z^3\ y + 1, z\ x^2 - t\right\}, \{t, z, y, x\}\right]$$

```
    Type:  List RegularTriangularSet( Integer, IndexedExponents
  OrderedVariableList [x,y,z,t], OrderedVariableList [x,y,z,t],
  NewSparseMultivariatePolynomial( Integer, OrderedVariableList
                                                   [x,y,z,t]))
```

We can see that the first decomposition is a subset of the second. So how can both be correct ?

Recall first that polynomials from a domain of the category `RecursivePolynomialCategory` are regarded as univariate polynomials in their main variable. For instance the second polynomial in the first set of each decomposition has main variable **y** and its initial (i.e. its leading coefficient w.r.t. its main variable) is **t z**.

Now let us explain how to read the second decomposition. Note that the non-constant initials of the first set are $t^4 - t$ and tz. Then the solutions described by this first set are the common zeros of its polynomials that do not cancel the polynomials $t^4 - t$ and tyz. Now the solutions of the input system **lp** satisfying these equations are described by the second and the third sets of the decomposition. Thus, in some sense, they can be considered as degenerated solutions. The solutions given by the first set are called the generic points of the system; they give the general form of the solutions. The first decomposition only provides these generic points. This latter decomposition is useful when

they are many degenerated solutions (which is sometimes hard to compute) and when one is only interested in general informations, like the dimension of the input system.

We can get the dimensions of each component of a decomposition as follows.

```
[coHeight(ts) for ts in lts]
```

$$[1, 0, 0]$$

<div align="right">

Type: List NonNegativeInteger
</div>

Thus the first set has dimension one. Indeed **t** can take any value, except **0** or any third root of **1**, whereas **z** is completely determined from **t**, **y** is given by **z** and **t**, and finally **x** is given by the other three variables. In the second and the third sets of the second decomposition the four variables are completely determined and thus these sets have dimension zero.

We give now the precise specifications of each decomposition. This assume some mathematical knowledge. However, for the non-expert user, the above explanations will be sufficient to understand the other features of the `RSEGSET` constructor.

The input system **lp** is decomposed in the sense of Kalkbrener as finitely many regular sets **T1,...,Ts** such that the radical ideal generated by **lp** is the intersection of the radicals of the saturated ideals of **T1,...,Ts**. In other words, the affine variety associated with **lp** is the union of the closures (w.r.t. Zarisky topology) of the regular-zeros sets of **T1,...,Ts**.

N. B. The prime ideals associated with the radical of the saturated ideal of a regular triangular set have all the same dimension; moreover these prime ideals can be given by characteristic sets with the same main variables. Thus a decomposition in the sense of Kalkbrener is unmixed dimensional. Then it can be viewed as a *lazy* decomposition into prime ideals (some of these prime ideals being merged into unmixed dimensional ideals).

Now we explain the other way of solving by means of regular triangular sets. The input system **lp** is decomposed in the sense of Lazard as finitely many regular triangular sets **T1,...,Ts** such that the affine variety associated with **lp** is the union of the regular-zeros sets of **T1,...,Ts**. Thus a decomposition in the sense of Lazard is also a decomposition in the sense of Kalkbrener; the converse is false as we have seen before.

When the input system has a finite number of solutions, both ways of solving provide similar decompositions as we shall see with this second example (Caprasse).

Define a polynomial system.

```
f1 := y**2*z+2*x*y*t-2*x-z
```

$$(2\ t\ y - 2)\ x + z\ y^2 - z$$

<div align="right">

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[x,y,z,t])

</div>

```
f2 := -x**3*z+ 4*x*y**2*z+ 4*x**2*y*t+ 2*y**3*t+ 4*x**2- 10*y**2+
4*x*z- 10*y*t+ 2
```

$$-z\ x^3 + (4\ t\ y + 4)\ x^2 + \left(4\ z\ y^2 + 4\ z\right)\ x + 2\ t\ y^3 - 10\ y^2 - 10\ t\ y + 2$$

<div align="right">

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[x,y,z,t])

</div>

```
f3 := 2*y*z*t+x*t**2-x-2*z
```

$$\left(t^2 - 1\right)\ x + 2\ t\ z\ y - 2\ z$$

<div align="right">

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[x,y,z,t])

</div>

```
f4 := -x*z**3+ 4*y*z**2*t+ 4*x*z*t**2+ 2*y*t**3+ 4*x*z+
4*z**2-10*y*t- 10*t**2+2
```

$$\left(-z^3 + \left(4\ t^2 + 4\right)\ z\right)\ x + \left(4\ t\ z^2 + 2\ t^3 - 10\ t\right)\ y + 4\ z^2 - 10\ t^2 + 2$$

<div align="right">

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[x,y,z,t])

</div>

```
lf := [f1, f2, f3, f4]
```

$$\left[(2\ t\ y - 2)\ x + z\ y^2 - z,\right.$$

$$-z\ x^3 + (4\ t\ y + 4)\ x^2 + \left(4\ z\ y^2 + 4\ z\right)\ x + 2\ t\ y^3 - 10\ y^2 - 10\ t\ y + 2,$$

$$\left(t^2 - 1\right)\ x + 2\ t\ z\ y - 2\ z,$$

$$\left.\left(-z^3 + \left(4\ t^2 + 4\right)\ z\right)\ x + \left(4\ t\ z^2 + 2\ t^3 - 10\ t\right)\ y + 4\ z^2 - 10\ t^2 + 2\right]$$

First of all, let us solve this system in the sense of Kalkbrener.

zeroSetSplit(lf)$T

$$\left[\left\{t^2 - 1, z^8 - 16\ z^6 + 256\ z^2 - 256, t\ y - 1, \left(z^3 - 8\ z\right)\ x - 8\ z^2 + 16\right\},\right.$$

$$\left\{3\ t^2 + 1, z^2 - 7\ t^2 - 1, y + t, x + z\right\},$$

$$\left\{t^8 - 10\ t^6 + 10\ t^2 - 1, z, \left(t^3 - 5\ t\right)\ y - 5\ t^2 + 1, x\right\},$$

$$\left.\left\{t^2 + 3, z^2 - 4, y + t, x - z\right\}\right]$$

And now in the sense of Lazard (or Wu and other authors).

lts2 := zeroSetSplit(lf,false)$T

$$\left[\left\{t^8 - 10\ t^6 + 10\ t^2 - 1, z, \left(t^3 - 5\ t\right)\ y - 5\ t^2 + 1, x\right\},\right.$$

$$\left\{t^2 - 1, z^8 - 16\ z^6 + 256\ z^2 - 256, t\ y - 1, \left(z^3 - 8\ z\right)\ x - 8\ z^2 + 16\right\},$$

$$\left\{3\ t^2 + 1, z^2 - 7\ t^2 - 1, y + t, x + z\right\},$$

$$\left.\left\{t^2 + 3, z^2 - 4, y + t, x - z\right\}\right]$$

Up to the ordering of the components, both decompositions are identical.

Let us check that each component has a finite number of solutions.

[coHeight(ts) for ts in lts2]

$$[0, 0, 0, 0]$$

Let us count the degrees of each component,

```
degrees := [degree(ts) for ts in lts2]
```

$$[8, 16, 4, 4]$$

Type: List NonNegativeInteger

and compute their sum.

```
reduce(+,degrees)
```

$$32$$

Type: PositiveInteger

We study now the options of the `zeroSetSplit` operation. As we have seen yet, there is an optional second argument which is a boolean value. If this value is `true` (this is the default) then the decomposition is computed in the sense of Kalkbrener, otherwise it is computed in the sense of Lazard.

There is a second boolean optional argument that can be used (in that case the first optional argument must be present). This second option allows you to get some information during the computations.

Therefore, we need to understand a little what is going on during the computations. An important feature of the algorithm is that the intermediate computations are managed in some sense like the processes of a Unix system. Indeed, each intermediate computation may generate other intermediate computations and the management of all these computations is a crucial task for the efficiency. Thus any intermediate computation may be suspended, killed or resumed, depending on algebraic considerations that determine priorities for these processes. The goal is of course to go as fast as possible towards the final decomposition which means to avoid as much as possible unnecessary computations.

To follow the computations, one needs to set to `true` the second argument. Then a lot of numbers and letters are displayed. Between a `[` and a `]` one has the state of the processes at a given time. Just after `[` one can see the number of processes. Then each process is represented by two numbers between `<` and `>`. A process consists of a list of polynomial **ps** and a triangular set **ts**; its goal is to compute the common zeros of **ps** that belong to the regular-zeros set of **ts**. After the processes, the number between pipes gives the total number of polynomials in all the sets **ps**. Finally, the number between braces gives the number of components of a decomposition that are already computed. This number may decrease.

Let us take a third example (Czapor-Geddes-Wang) to see how this information is displayed.

Define a polynomial system.

```
u :  R := 2
```

$$2$$

```
q1 := 2*(u-1)**2+ 2*(x-z*x+z**2)+ y**2*(x-1)**2- 2*u*x+
2*y*t*(1-x)*(x-z)+ 2*u*z*t*(t-y)+ u**2*t**2*(1-2*z)+
2*u*t**2*(z-x)+ 2*u*t*y*(z-1)+ 2*u*z*x*(y+1)+
(u**2-2*u)*z**2*t**2+ 2*u**2*z**2+ 4*u*(1-u)*z+ t**2*(z-x)**2
```

$$\left(y^2 - 2\ t\ y + t^2\right)\ x^2 +$$

$$\left(-2\ y^2 + ((2\ t + 4)\ z + 2\ t)\ y + \left(-2\ t^2 + 2\right)\ z - 4\ t^2 - 2\right)\ x +$$

$$y^2 + (-2\ t\ z - 4\ t)\ y + \left(t^2 + 10\right)\ z^2 - 8\ z + 4\ t^2 + 2$$

```
                                              Type:
            NewSparseMultivariatePolynomial(Integer,OrderedVariableList
                                                          [x,y,z,t])
```

```
q2 := t*(2*z+1)*(x-z)+ y*(z+2)*(1-x)+ u*(u-2)*t+ u*(1-2*u)*z*t+
u*y*(x+u-z*x-1)+ u*(u+1)*z**2*t
```

$$(-3\ z\ y + 2\ t\ z + t)\ x + (z + 4)\ y + 4\ t\ z^2 - 7\ t\ z$$

```
                                              Type:
            NewSparseMultivariatePolynomial(Integer,OrderedVariableList
                                                          [x,y,z,t])
```

```
q3 := -u**2*(z-1)**2+ 2*z*(z-x)-2*(x-1)
```

$$(-2\ z - 2)\ x - 2\ z^2 + 8\ z - 2$$

```
                                              Type:
            NewSparseMultivariatePolynomial(Integer,OrderedVariableList
                                                          [x,y,z,t])
```

```
q4 := u**2+4*(z-x**2)+3*y**2*(x-1)**2- 3*t**2*(z-x)**2
+3*u**2*t**2*(z-1)**2+u**2*z*(z-2)+6*u*t*y*(z+x+z*x-1)
```

$$\left(3\ y^2 - 3\ t^2 - 4\right)\ x^2 + \left(-6\ y^2 + (12\ t\ z + 12\ t)\ y + 6\ t^2\ z\right)\ x + 3\ y^2 +$$

$$(12\ t\ z - 12\ t)\ y + \left(9\ t^2 + 4\right)\ z^2 + \left(-24\ t^2 - 4\right)\ z + 12\ t^2 + 4$$

<div align="right">

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[x,y,z,t])

</div>

```
lq := [q1, q2, q3, q4]
```

$$\Big[\left(y^2 - 2\ t\ y + t^2\right)\ x^2 +$$

$$\left(-2\ y^2 + ((2\ t + 4)\ z + 2\ t)\ y + \left(-2\ t^2 + 2\right)\ z - 4\ t^2 - 2\right)\ x + y^2 +$$

$$(-2\ t\ z - 4\ t)\ y + \left(t^2 + 10\right)\ z^2 - 8\ z + 4\ t^2 + 2,$$

$$(-3\ z\ y + 2\ t\ z + t)\ x + (z + 4)\ y + 4\ t\ z^2 - 7\ t\ z,$$

$$(-2\ z - 2)\ x - 2\ z^2 + 8\ z - 2, \left(3\ y^2 - 3\ t^2 - 4\right)\ x^2 +$$

$$\left(-6\ y^2 + (12\ t\ z + 12\ t)\ y + 6\ t^2\ z\right)\ x + 3\ y^2 +$$

$$(12\ t\ z - 12\ t)\ y + \left(9\ t^2 + 4\right)\ z^2 + \left(-24\ t^2 - 4\right)\ z + 12\ t^2 + 4\Big]$$

<div align="right">

Type: List
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[x,y,z,t])

</div>

Let us try the information option. N.B. The timing should be between 1 and
10 minutes, depending on your machine.

```
zeroSetSplit(lq,true,true)$T
```

```
[1 <4,0> -> |4|; {0}]W[2 <5,0>,<3,1> -> |8|; {0}][2 <4,1>,<3,1> -> |7|;
{0}][1 <3,1> -> |3|; {0}]G[2 <4,1>,<4,1> -> |8|; {0}]W[3 <5,1>,<4,1>,
<3,2> -> |12|; {0}]GI[3 <4,2>,<4,1>,<3,2> -> |11|; {0}]GWw[3 <4,1>,
<3,2>,<5,2> -> |12|; {0}][3 <3,2>,<3,2>,<5,2> -> |11|; {0}]GIwWWWw
[4 <3,2>,<4,2>,<5,2>,<2,3> -> |14|; {0}][4 <2,2>,<4,2>,<5,2>,<2,3> ->
|13|; {0}]Gwww[5 <3,2>,<3,2>,<4,2>,<5,2>,<2,3> -> |17|; {0}]Gwwwwww
[8 <3,2>,<4,2>,<4,2>,<4,2>,<4,2>,<4,2>,<5,2>,<2,3> -> |30|; {0}]Gwwwwww
[8 <4,2>,<4,2>,<4,2>,<4,2>,<4,2>,<4,2>,<5,2>,<2,3> -> |31|; {0}][8
<3,3>,<4,2>,<4,2>,<4,2>,<4,2>,<4,2>,<5,2>,<2,3> -> |30|; {0}][8 <2,3>,
```

<4,2>,<4,2>,<4,2>,<4,2>,<4,2>,<5,2>,<2,3> -> |29|; {0}][8 <1,3>,<4,2>,
<4,2>,<4,2>,<4,2>,<4,2>,<5,2>,<2,3> -> |28|; {0}][7 <4,2>,<4,2>,<4,2>,
<4,2>,<4,2>,<5,2>,<2,3> -> |27|; {0}][6 <4,2>,<4,2>,<4,2>,<4,2>,<5,2>,
<2,3> -> |23|; {0}][5 <4,2>,<4,2>,<4,2>,<5,2>,<2,3> -> |19|; {0}]
GIGIWwww[6 <5,2>,<4,2>,<4,2>,<5,2>,<3,3>,<2,3> -> |23|; {0}][6 <4,3>,
<4,2>,<4,2>,<5,2>,<3,3>,<2,3> -> |22|; {0}]GIGI[6 <3,4>,<4,2>,<4,2>,
<5,2>,<3,3>,<2,3> -> |21|; {0}][6 <2,4>,<4,2>,<4,2>,<5,2>,<3,3>,<2,3>
-> |20|; {0}]GGG[5 <4,2>,<4,2>,<5,2>,<3,3>,<2,3> -> |18|; {0}]GIGIWwwwW
[6 <5,2>,<4,2>,<5,2>,<3,3>,<3,3>,<2,3> -> |22|; {0}][6 <4,3>,<4,2>,
<5,2>,<3,3>,<3,3>,<2,3> -> |21|; {0}]GIwwWwWWWWWWWWwWWWWwwww[8 <4,2>,

<5,2>,<3,3>,<3,3>,<4,3>,<2,3>,<3,4>,<3,4> -> |27|; {0}][8 <3,3>,<5,2>,
<3,3>,<3,3>,<4,3>,<2,3>,<3,4>,<3,4> -> |26|; {0}][8 <2,3>,<5,2>,<3,3>,
<3,3>,<4,3>,<2,3>,<3,4>,<3,4> -> |25|; {0}]Gwwwwwwwwwwwwwwwwwwww[9
<5,2>,<3,3>,<3,3>,<4,3>,<3,3>,<3,3>,<2,3>,<3,4>,<3,4> -> |29|; {0}]
GI[9 <4,3>,<3,3>,<3,3>,<4,3>,<3,3>,<3,3>,<2,3>,<3,4>,<3,4> -> |28|;
{0}][9 <3,3>,<3,3>,<3,3>,<4,3>,<3,3>,<3,3>,<2,3>,<3,4>,<3,4> -> |27|;
{0}][9 <2,3>,<3,3>,<3,3>,<4,3>,<3,3>,<3,3>,<2,3>,<3,4>,<3,4> -> |26|;
{0}]GGwwwwwwwwwwwwwWWwwwwwwww[11 <3,3>,<3,3>,<3,3>,<3,3>,<4,3>,<2,3>,
<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |33|; {0}][11 <2,3>,<3,3>,<3,3>,<3,3>,
<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |32|; {0}][11 <1,3>,<3,3>,

<3,3>,<3,3>,<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |31|; {0}]
GGGwwwwwwwwwwwww[12 <2,3>,<2,3>,<3,3>,<3,3>,<4,3>,<3,3>,<2,3>,<3,3>,
<3,3>,<3,3>,<3,4>,<3,4> -> |34|; {0}]GGwwwwwwwwwwwww[13 <3,3>,<2,3>,
<3,3>,<3,3>,<3,3>,<3,3>,<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> ->
|38|; {0}]Gwwwwwwwwwwwwww[13 <2,3>,<3,3>,<4,3>,<3,3>,<4,3>,<3,3>,<3,3>,
<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |39|; {0}]GGGwwwwwwwwwwwww[15
<3,3>,<4,3>,<3,3>,<3,3>,<3,3>,<3,3>,<3,3>,<3,3>,<4,3>,<2,3>,<3,3>,<3,3>,
<3,3>,<3,4>,<3,4> -> |46|; {0}][14 <4,3>,<3,3>,<3,3>,<3,3>,<3,3>,<3,3>,
<3,3>,<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |43|; {0}]GIGGGGIGGI
[14 <3,4>,<3,3>,<3,3>,<3,3>,<3,3>,<3,3>,<3,3>,<4,3>,<2,3>,<3,3>,<3,3>,

<3,3>,<3,4>,<3,4> -> |42|; {0}]GGG[14 <2,4>,<3,3>,<3,3>,<3,3>,<3,3>,
<3,3>,<3,3>,<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |41|; {0}]
[14 <1,4>,<3,3>,<3,3>,<3,3>,<3,3>,<3,3>,<3,3>,<4,3>,<2,3>,<3,3>,<3,3>,
<3,3>,<3,4>,<3,4> -> |40|; {0}]GGG[13 <3,3>,<3,3>,<3,3>,<3,3>,<3,3>,
<3,3>,<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |39|; {0}]
Gwwwwwwwwwwwww[15 <3,3>,<3,3>,<4,3>,<4,3>,<4,3>,<3,3>,<3,3>,<4,3>,
<3,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |48|; {0}]Gwwwwwwwwwwwww
[15 <4,3>,<4,3>,<3,3>,<4,3>,<4,3>,<3,3>,<4,3>,<3,3>,<3,3>,<2,3>,<3,3>,
<3,3>,<3,3>,<3,4>,<3,4> -> |49|; {0}]GIGI[15 <3,4>,<4,3>,<3,3>,<4,3>,
<4,3>,<3,3>,<4,3>,<3,3>,<3,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> ->

|48|; {0}]G[14 <4,3>,<3,3>,<4,3>,<4,3>,<3,3>,<4,3>,<3,3>,<3,3>,<2,3>,

```
<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |45|; {0}][13 <3,3>,<4,3>,<4,3>,
<3,3>,<4,3>,<3,3>,<3,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |41|;
{0}]Gwwwwwwwwwwwww[13 <4,3>,<4,3>,<4,3>,<3,3>,<3,3>,<4,3>,<3,3>,<2,3>,
<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |42|; {0}]GIGGGGIGGI[13 <3,4>,<4,3>,
<4,3>,<3,3>,<3,3>,<4,3>,<3,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> ->
|41|; {0}]GGGGGGGG[13 <2,4>,<4,3>,<4,3>,<3,3>,<3,3>,<4,3>,<3,3>,<2,3>,
<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |40|; {0}][13 <1,4>,<4,3>,<4,3>,<3,3>,
<3,3>,<4,3>,<3,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |39|; {0}]
[13 <0,4>,<4,3>,<4,3>,<3,3>,<3,3>,<4,3>,<3,3>,<2,3>,<3,3>,<3,3>,<3,3>,

<3,4>,<3,4> -> |38|; {0}][12 <4,3>,<4,3>,<3,3>,<3,3>,<4,3>,<3,3>,
<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |38|; {1}][11 <4,3>,<3,3>,
<3,3>,<4,3>,<3,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |34|; {1}]
[10 <3,3>,<3,3>,<4,3>,<3,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> ->
|30|; {1}][10 <2,3>,<3,3>,<4,3>,<3,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,
<3,4> -> |29|; {1}]GGGwwwwwwwwwwwww[11 <3,3>,<3,3>,<4,3>,<3,3>,
<3,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |33|; {1}]
GGGwwwwwwwwwwwww[12 <4,3>,<3,3>,<4,3>,<3,3>,<3,3>,<4,3>,
<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |38|; {1}]Gwwwwwwwwwwwww
[12 <3,3>,<4,3>,<5,3>,<3,3>,<4,3>,<3,3>,<2,3>,<3,3>,<3,3>,<3,3>,

<3,4>,<3,4> -> |39|; {1}]GGwwwwwwwwwwwww[13 <5,3>,<4,3>,<4,3>,
<4,3>,<3,3>,<3,3>,<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> ->
|44|; {1}]GIGGGGIGGIW[13 <4,4>,<4,3>,<4,3>,<4,3>,<3,3>,<3,3>,
<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |43|; {1}]GGW[13
<3,4>,<4,3>,<4,3>,<4,3>,<3,3>,<3,3>,<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,
<3,4>,<3,4> -> |42|; {1}]GGG[12 <4,3>,<4,3>,<4,3>,<3,3>,<3,3>,<4,3>,
<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |39|; {1}]Gwwwwwwwwwwwww[12
<4,3>,<4,3>,<5,3>,<3,3>,<4,3>,<3,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,
<3,4> -> |40|; {1}]Gwwwwwwwwwwwww[13 <5,3>,<5,3>,<4,3>,<5,3>,<3,3>,
<3,3>,<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |46|; {1}]GIGIW

[13 <4,4>,<5,3>,<4,3>,<5,3>,<3,3>,<3,3>,<4,3>,<2,3>,<3,3>,<3,3>,
<3,3>,<3,4>,<3,4> -> |45|; {1}][13 <3,4>,<5,3>,<4,3>,<5,3>,<3,3>,
<3,3>,<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |44|; {1}][13
<2,4>,<5,3>,<4,3>,<5,3>,<3,3>,<3,3>,<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,
<3,4>,<3,4> -> |43|; {1}]GG[12 <5,3>,<4,3>,<5,3>,<3,3>,<3,3>,<4,3>,
<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |41|; {1}]GIGGGGIGGIW[12
<4,4>,<4,3>,<5,3>,<3,3>,<3,3>,<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,
<3,4> -> |40|; {1}]GGGGGGW[12 <3,4>,<4,3>,<5,3>,<3,3>,<3,3>,<4,3>,
<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |39|; {1}][12 <2,4>,<4,3>,
<5,3>,<3,3>,<3,3>,<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |38|;

{1}][12 <1,4>,<4,3>,<5,3>,<3,3>,<3,3>,<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,
<3,4>,<3,4> -> |37|; {1}]GGG[11 <4,3>,<5,3>,<3,3>,<3,3>,<4,3>,<2,3>,
```

```
<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |36|; {1}][10 <5,3>,<3,3>,<3,3>,
<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |32|; {1}][9 <3,3>,
<3,3>,<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |27|; {1}]W[9
<2,4>,<3,3>,<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |26|; {1}]
[9 <1,4>,<3,3>,<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |25|;
{1}][8 <3,3>,<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |24|; {1}]
W[8 <2,4>,<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |23|; {1}][8
<1,4>,<4,3>,<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |22|; {1}][7 <4,3>,



<2,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |21|; {1}]w[7 <3,4>,<2,3>,
<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |20|; {1}][7 <2,4>,<2,3>,<3,3>,
<3,3>,<3,3>,<3,4>,<3,4> -> |19|; {1}][7 <1,4>,<2,3>,<3,3>,<3,3>,
<3,3>,<3,4>,<3,4> -> |18|; {1}][6 <2,3>,<3,3>,<3,3>,<3,3>,<3,4>,
<3,4> -> |17|; {1}]GGwwwwww[7 <3,3>,<3,3>,<3,3>,<3,3>,<3,3>,<3,4>,
<3,4> -> |21|; {1}]GIW[7 <2,4>,<3,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4>
-> |20|; {1}]GG[6 <3,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |18|; {1}]
Gwwwwww[7 <4,3>,<4,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |23|; {1}]
GIW[7 <3,4>,<4,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |22|; {1}][6
<4,3>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |19|; {1}]GIW[6 <3,4>,<3,3>,



<3,3>,<3,3>,<3,4>,<3,4> -> |18|; {1}]GGW[6 <2,4>,<3,3>,<3,3>,<3,3>,
<3,4>,<3,4> -> |17|; {1}][6 <1,4>,<3,3>,<3,3>,<3,3>,<3,4>,<3,4> ->
|16|; {1}]GGG[5 <3,3>,<3,3>,<3,3>,<3,4>,<3,4> -> |15|; {1}]GIW[5
<2,4>,<3,3>,<3,3>,<3,4>,<3,4> -> |14|; {1}]GG[4 <3,3>,<3,3>,<3,4>,
<3,4> -> |12|; {1}][3 <3,3>,<3,4>,<3,4> -> |9|; {1}]W[3 <2,4>,<3,4>,
<3,4> -> |8|; {1}][3 <1,4>,<3,4>,<3,4> -> |7|; {1}]G[2 <3,4>,<3,4>
-> |6|; {1}]G[1 <3,4> -> |3|; {1}][1 <2,4> -> |2|; {1}][1 <1,4> ->
|1|; {1}]
```

```
   *** QCMPACK Statistics ***
      Table      size:  36
      Entries reused:  255

   *** REGSETGCD: Gcd Statistics ***
      Table      size:  125
      Entries reused:  0

   *** REGSETGCD: Inv Set Statistics ***
      Table      size:  30
      Entries reused:  0
```

$\Big[\Big\{960725655771966\ t^{24} + 386820897948702\ t^{23}+$
$8906817198608181\ t^{22} + 2704966893949428\ t^{21}+$
$37304033340228264\ t^{20} + 7924782817170207\ t^{19}+$
$93126799040354990\ t^{18} + 13101273653130910\ t^{17}+$
$156146250424711858\ t^{16} + 16626490957259119\ t^{15}+$
$190699288479805763\ t^{14} + 24339173367625275\ t^{13}+$
$180532313014960135\ t^{12} + 35288089030975378\ t^{11}+$

$135054975747656285\ t^{10} + 34733736952488540\ t^{9}\ +$
$75947600354493972\ t^{8} + 19772555692457088\ t^{7}+$
$28871558573755428\ t^{6} + 5576152439081664\ t^{5}+$
$6321711820352976\ t^{4} + 438314209312320\ t^{3}+$
$581105748367008\ t^{2} - 60254467992576\ t+$
$1449115951104,$

$\big(266042108694913023855152657370520823616684741813728918$57784\ t^{23}+$
$4431043784246860860672948995282966642386935568550177352$65295\ t^{22}+$
$2790783932867012346791413423589883271553213058295470903$10242\ t^{21}+$
$3390276361413232465107617176615543054620626391823613392$185226\ t^{20}+$
$9414781795035405755541986452203528037197931964738138374$34129\ t^{19}+$
$1154785519467947524221169674967394935258574767418432098$8144390\ t^{18}+$
$1343609566765597789881701656699413216467215660333356417$241432\ t^{17}+$
$2323381386814787350393355161717564085989910298780066356$6699334\ t^{16}+$
$8695740205376723369508454405087907408509313364849835733$86433\ t^{15}+$
$3156155430587693487541946148696992655424175006510346082$0476969\ t^{14}+$

$1271400990287717487442065952547731879554823889855386072$264931\ t^{13}+$
$3194508991386373604480252696407954019833704955050329582$5160523\ t^{12}+$
$3738735704288144509871371560232845884439102270778010470$931960\ t^{11}+$
$2529399751239141202614460143577113158756190553299204569$2885927\ t^{10}+$
$5210239009846067123469262799870052773410471135950175008$046524\ t^{9}+$
$1508388798693029716625987056860827042740318760623871349$1129188\ t^{8}+$
$3522087234692930126383686270775779553481769125670839075$109000\ t^{7}+$
$6079945200395681013086533792568886491101244247440034969$288588\ t^{6}+$
$1090634852433900888199913756247986023196987723469934933$603680\ t^{5}+$
$1405819430871907102294432537538335402102838994019667487$458352\ t^{4}+$

$8807152795032045007253667126550774887834782888493360520$2432\ t^{3}+$
$1358824894336409332297811771559777680160657654823786571$29440\ t^{2}-$
$1395728344288226223055989460740031408251669074997564652$0320\ t+$
$3346376929731892992772583257093084725921171128557497139$13920\big)\ z+$
$8567175484043952879756725964506833932149637101090521164$936\ t^{23}+$
$1497923928642017918457083740327289424987975192516672509$45721\ t^{22}+$
$7725837178364582215741086158215976413812300307419037402$1550\ t^{21}+$
$1108862254126854214498918940708612211184560556764334742$191654\ t^{20}+$
$2132504944606788652197744801068260537838157896215017326$72327\ t^{19}+$

$3668929075160666195729177894178343514501987898410131431699882\ t^{18}+$

$1713889064710018728794901243687482363147654590395567820048872\ t^{17}+$

$7192430746914602166660233477331022483144921771645523139658986\ t^{16}-$

$1287986746896900728128799656330902919596631431084373624533385\ t^{15}+$

$9553010858341425909306423132921134040856028790803526430270671\ t^{14}-$

$1329609624567549287453868764630043782465845870914444109660 3\ t^{13}+$

$947580680581414532638308551832533310688169056864427496486441 3\ t^{12}+$

$8032346879251334588616598556640849276062987947998562655393 36\ t^{11}+$

$7338202759292865165994622349207516400662174302614595173333825\ t^{10}+$

$1308004628480367351164369613111971668880538855640917200187108\ t^{9}+$

$4268059455741255498880229598973705747098216067697754352634748\ t^{8}+$

$8928935268585140957913187759040933001030456015144706135806 00\ t^{7}+$

$1679152575460683956631925852181341501981598137465328797013652\ t^{6}+$

$2697574157679229803789671541433578355441131582805914080439 36\ t^{5}+$

$3809515278646575290335808298012827240813453726802029201982 24\ t^{4}+$

$1978554529422849503299882693760134113272503533945291328665 6\ t^{3}+$

$3647741205738478294236663530339663776330392817493507917852 8\ t^{2}-$

$3722212879279038648713080422224976273210890229485838670848\ t+$

$8907972485311434836123063448401386202472859990687410585 6,$

$$\left(3\ z^{3}-11\ z^{2}+8\ z+4\right)\ y+2\ t\ z^{3}+4\ t\ z^{2}-5\ t\ z-t,:$$

$$(z+1)\ x+z^{2}-4\ z+1\}]$$

```
Type:  List RegularTriangularSet( Integer, IndexedExponents
OrderedVariableList [x,y,z,t], OrderedVariableList [x,y,z,t],
  NewSparseMultivariatePolynomial(Integer, OrderedVariableList
                                                  [x,y,z,t]))
```

Between a sequence of processes, thus between a] and a [you can see capital letters W, G, I and lower case letters i, w. Each time a capital letter appears a non-trivial computation has be performed and its result is put in a hash-table. Each time a lower case letter appears a needed result has been found in an hash-table. The use of these hash-tables generally speed up the computations. However, on very large systems, it may happen that these hash-tables become too big to be handle by your AXIOM configuration. Then in these exceptional cases, you may prefer getting a result (even if it takes a long time) than getting nothing. Hence you need to know how to prevent the RSEGSET constructor from using these hash-tables. In that case you will be using the zeroSetSplit with five arguments. The first one is the input system **lp** as above. The second one is a boolean value **hash?** which is true iff you want to use hash-tables. The third one is boolean value **clos?** which is true iff you want to solve your system in the sense of Kalkbrener, the other way remaining that of Lazard. The fourth argument is boolean value **info?** which is true iff you want to display information during the computations. The last one is boolean value

prep? which is `true` iff you want to use some heuristics that are performed on the input system before starting the real algorithm. The value of this flag is `true` when you are using `zeroSetSplit` with less than five arguments. Note that there is no available signature for `zeroSetSplit` with four arguments.

We finish this section by some remarks about both ways of solving, in the sense of Kalkbrener or in the sense of Lazard. For problems with a finite number of solutions, there are theoretically equivalent and the resulting decompositions are identical, up to the ordering of the components. However, when solving in the sense of Lazard, the algorithm behaves differently. In that case, it becomes more incremental than in the sense of Kalkbrener. That means the polynomials of the input system are considered one after another whereas in the sense of Kalkbrener the input system is treated more globally.

This makes an important difference in positive dimension. Indeed when solving in the sense of Kalkbrener, the *Primeidealkettensatz* of Krull is used. That means any regular triangular containing more polynomials than the input system can be deleted. This is not possible when solving in the sense of Lazard. This explains why Kalkbrener's decompositions usually contain less components than those of Lazard. However, it may happen with some examples that the incremental process (that cannot be used when solving in the sense of Kalkbrener) provide a more efficient way of solving than the global one even if the *Primeidealkettensatz* is used. Thus just try both, with the various options, before concluding that you cannot solve your favorite system with `zeroSetSplit`. There exist more options at the development level that are not currently available in this public version.

9.68   RomanNumeral

The Roman numeral package was added to Axiom in MCMLXXXVI for use in denoting higher order derivatives.

For example, let `f` be a symbolic operator.

```
f := operator 'f
```

$$f$$

                                        Type: BasicOperator

This is the seventh derivative of `f` with respect to `x`.

```
D(f x,x,7)
```

$$f^{(vii)}(x)$$

You can have integers printed as Roman numerals by declaring variables to be of type `RomanNumeral` (abbreviation `ROMAN`).

```
a := roman(1978 - 1965)
```

$$XIII$$

Type: RomanNumeral

This package now has a small but devoted group of followers that claim this domain has shown its efficacy in many other contexts. They claim that Roman numerals are every bit as useful as ordinary integers.

In a sense, they are correct, because Roman numerals form a ring and you can therefore construct polynomials with Roman numeral coefficients, matrices over Roman numerals, etc..

```
x :  UTS(ROMAN,'x,0) := x
```

$$x$$

Type: UnivariateTaylorSeries(RomanNumeral,x,0)

Was Fibonacci Italian or ROMAN?

```
recip(1 - x - x**2)
```

$$I + x + II\ x^2 + III\ x^3 + V\ x^4 + VIII\ x^5 + XIII\ x^6 + XXI\ x^7 +$$

$$XXXIV\ x^8 + LV\ x^9 + LXXXIX\ x^{10} + O\left(x^{11}\right)$$

Type: Union(UnivariateTaylorSeries(RomanNumeral,x,0),...)

You can also construct fractions with Roman numeral numerators and denominators, as this matrix Hilberticus illustrates.

```
m :  MATRIX FRAC ROMAN
```

Void

```
m := matrix [ [1/(i + j) for i in 1..3] for j in 1..3]
```

$$
\begin{bmatrix}
\frac{I}{II} & \frac{I}{III} & \frac{I}{IV} \\
\frac{I}{III} & \frac{I}{IV} & \frac{I}{V} \\
\frac{I}{IV} & \frac{I}{V} & \frac{I}{VI}
\end{bmatrix}
$$

<div align="right">Type: Matrix Fraction RomanNumeral</div>

Note that the inverse of the matrix has integral `ROMAN` entries.

```
inverse m
```

$$
\begin{bmatrix}
LXXII & -CCXL & CLXXX \\
-CCXL & CM & -DCCXX \\
CLXXX & -DCCXX & DC
\end{bmatrix}
$$

<div align="right">Type: Union(Matrix Fraction RomanNumeral,...)</div>

Unfortunately, the spoil-sports say that the fun stops when the numbers get big—mostly because the Romans didn't establish conventions about representing very large numbers.

```
y := factorial 10
```

<div align="center">3628800</div>

<div align="right">Type: PositiveInteger</div>

You work it out!

```
roman y
```

<div align="center">((((I))))((((I))))((((I))))(((I)))(((I)))(((I)))(((I)))
(((I)))(((I)))((I))((I))MMMMMMMMDCCC</div>

<div align="right">Type: RomanNumeral</div>

Issue the system command `)show RomanNumeral` to display the full list of operations defined by `RomanNumeral`.

9.69 Segment

The `Segment` domain provides a generalized interval type.

Segments are created using the `..` construct by indicating the (included) end points.

```
s := 3..10
```

$$3..10$$

<div align="right">Type: Segment PositiveInteger</div>

The first end point is called the **lo** and the second is called **hi**.

```
lo s
```

$$3$$

<div align="right">Type: PositiveInteger</div>

These names are used even though the end points might belong to an unordered set.

```
hi s
```

$$10$$

<div align="right">Type: PositiveInteger</div>

In addition to the end points, each segment has an integer "increment." An increment can be specified using the "`by`" construct.

```
t := 10..3 by -2
```

$$10..3\text{by} - 2$$

<div align="right">Type: Segment PositiveInteger</div>

This part can be obtained using the **incr** function.

```
incr s
```

$$1$$

<div align="right">Type: PositiveInteger</div>

Unless otherwise specified, the increment is `1`.

```
incr t
```

$$-2$$

<div align="right">Type: Integer</div>

A single value can be converted to a segment with equal end points. This happens if segments and single values are mixed in a list.

```
l := [1..3, 5, 9, 15..11 by -1]
```

$$[1..3, 5..5, 9..9, 15..11\text{by} -1]$$

<div align="right">Type: List Segment PositiveInteger</div>

If the underlying type is an ordered ring, it is possible to perform additional operations. The **expand** operation creates a list of points in a segment.

```
expand s
```

$$[3, 4, 5, 6, 7, 8, 9, 10]$$

<div align="right">Type: List Integer</div>

If `k > 0`, then `expand(l..h by k)` creates the list `[l, l+k, ..., lN]` where `lN <= h < lN+k`. If `k < 0`, then `lN >= h > lN+k`.

```
expand t
```

$$[10, 8, 6, 4]$$

<div align="right">Type: List Integer</div>

It is also possible to expand a list of segments. This is equivalent to appending lists obtained by expanding each segment individually.

```
expand l
```

$$[1, 2, 3, 5, 9, 15, 14, 13, 12, 11]$$

<div align="right">Type: List Integer</div>

For more information on related topics, see 9.70 on page 785 and 9.84 on page 835.

9.70 SegmentBinding

The `SegmentBinding` type is used to indicate a range for a named symbol. First give the symbol, then an `=` and finally a segment of values.

```
x = a..b
```

$$x = a..b$$

<div align="right">Type: SegmentBinding Symbol</div>

This is used to provide a convenient syntax for arguments to certain operations.

```
sum(i**2, i = 0..n)
```

$$\frac{2\ n^3 + 3\ n^2 + n}{6}$$

<div align="right">Type: Fraction Polynomial Integer</div>

```
draw(x**2, x = -2..2)
```

The left-hand side must be of type `Symbol` but the right-hand side can be a segment over any type.

```
sb := y = 1/2..3/2
```

$$y = \left(\frac{1}{2}\right)..\left(\frac{3}{2}\right)$$

<div align="right">Type: SegmentBinding Fraction Integer</div>

The left- and right-hand sides can be obtained using the **variable** and **segment** operations.

```
variable(sb)
```

$$y$$

<div align="right">Type: Symbol</div>

```
segment(sb)
```

$$\left(\frac{1}{2}\right)..\left(\frac{3}{2}\right)$$

<div align="right">Type: Segment Fraction Integer</div>

For more information on related topics, see 9.69 on page 783 and 9.84 on page 835.

9.71   Set

The `Set` domain allows one to represent explicit finite sets of values. These are similar to lists, but duplicate elements are not allowed.

Sets can be created by giving a fixed set of values ...

```
s := set [x**2-1, y**2-1, z**2-1]
```

$$\left\{x^2 - 1, y^2 - 1, z^2 - 1\right\}$$

<div align="right">Type: Set Polynomial Integer</div>

or by using a collect form, just as for lists. In either case, the set is formed from a finite collection of values.

```
t := set [x**i - i+1 for i in 2..10 | prime?  i]
```

$$\left\{x^2 - 1, x^3 - 2, x^5 - 4, x^7 - 6\right\}$$

<div align="right">Type: Set Polynomial Integer</div>

The basic operations on sets are **intersect**, **union**, **difference**, and **symmetricDifference**.

```
i := intersect(s,t)
```

$$\left\{ x^2 - 1 \right\}$$

<div align="right">Type: Set Polynomial Integer</div>

u := union(s,t)

$$\left\{ x^2 - 1, x^3 - 2, x^5 - 4, x^7 - 6, y^2 - 1, z^2 - 1 \right\}$$

<div align="right">Type: Set Polynomial Integer</div>

The set `difference(s,t)` contains those members of `s` which are not in `t`.

difference(s,t)

$$\left\{ y^2 - 1, z^2 - 1 \right\}$$

<div align="right">Type: Set Polynomial Integer</div>

The set `symmetricDifference(s,t)` contains those elements which are in `s` or `t` but not in both.

symmetricDifference(s,t)

$$\left\{ x^3 - 2, x^5 - 4, x^7 - 6, y^2 - 1, z^2 - 1 \right\}$$

<div align="right">Type: Set Polynomial Integer</div>

Set membership is tested using the **member?** operation.

member?(y, s)

<div align="center">false</div>

<div align="right">Type: Boolean</div>

member?((y+1)*(y-1), s)

<div align="center">true</div>

<div align="right">Type: Boolean</div>

The **subset?** function determines whether one set is a subset of another.

```
subset?(i, s)
```

$$true$$

<div align="right">Type: Boolean</div>

```
subset?(u, s)
```

$$false$$

<div align="right">Type: Boolean</div>

When the base type is finite, the absolute complement of a set is defined. This finds the set of all multiplicative generators of `PrimeField 11`—the integers mod `11`.

```
gs := set [g for i in 1..11 | primitive?(g := i::PF 11)]
```

$$\{2, 6, 7, 8\}$$

<div align="right">Type: Set PrimeField 11</div>

The following values are not generators.

```
complement gs
```

$$\{1, 3, 4, 5, 9, 10, 0\}$$

<div align="right">Type: Set PrimeField 11</div>

Often the members of a set are computed individually; in addition, values can be inserted or removed from a set over the course of a computation.

There are two ways to do this:

```
a := set [i**2 for i in 1..5]
```

$$\{1, 4, 9, 16, 25\}$$

<div align="right">Type: Set PositiveInteger</div>

One is to view a set as a data structure and to apply updating operations.

```
insert!(32, a)
```

$$\{1, 4, 9, 16, 25, 32\}$$

<div align="right">Type: Set PositiveInteger</div>

```
remove!(25, a)
```

$$\{1, 4, 9, 16, 32\}$$

<div align="right">Type: Set PositiveInteger</div>

```
a
```

$$\{1, 4, 9, 16, 32\}$$

<div align="right">Type: Set PositiveInteger</div>

The other way is to view a set as a mathematical entity and to create new sets from old.

```
b := b0 := set [i**2 for i in 1..5]
```

$$\{1, 4, 9, 16, 25\}$$

<div align="right">Type: Set PositiveInteger</div>

```
b := union(b, 32)
```

$$\{1, 4, 9, 16, 25, 32\}$$

<div align="right">Type: Set PositiveInteger</div>

```
b := difference(b, 25)
```

$$\{1, 4, 9, 16, 32\}$$

<div align="right">Type: Set PositiveInteger</div>

```
b0
```

$$\{1, 4, 9, 16, 25\}$$

<div align="right">Type: Set PositiveInteger</div>

For more information about lists, see 9.47 on page 675.

9.72   SingleInteger

The `SingleInteger` domain is intended to provide support in Axiom for machine integer arithmetic. It is generally much faster than (bignum) `Integer` arithmetic but suffers from a limited range of values. Since Axiom can be implemented on top of various dialects of Lisp, the actual representation of small integers may not correspond exactly to the host machines integer representation.

In the CCL implementation of AXIOM (Release 2.1 onwards) the underlying representation of `SingleInteger` is the same as `Integer`. The underlying Lisp primitives treat machine-word sized computations specially.

You can discover the minimum and maximum values in your implementation by using **min** and **max**.

```
min()$SingleInteger
```

$$-134217728$$

<div align="right">Type: SingleInteger</div>

```
max()$SingleInteger
```

$$134217727$$

<div align="right">Type: SingleInteger</div>

To avoid confusion with `Integer`, which is the default type for integers, you usually need to work with declared variables (**??** on page **??** in Section **??** on page **??**) . . .

```
a := 1234 ::  SingleInteger
```

$$1234$$

<div align="right">Type: SingleInteger</div>

or use package calling (**??** on page **??** in Section **??** on page **??**).

```
b := 124$SingleInteger
```

$$124$$

<div align="right">Type: SingleInteger</div>

You can add, multiply and subtract `SingleInteger` objects, and ask for the greatest common divisor (`gcd`).

`gcd(a,b)`

$$2$$

<div align="right">Type: SingleInteger</div>

The least common multiple (`lcm`) is also available.

`lcm(a,b)`

$$76508$$

<div align="right">Type: SingleInteger</div>

Operations **mulmod**, **addmod**, **submod**, and **invmod** are similar—they provide arithmetic modulo a given small integer. Here is $5 * 6 \mathbf{mod} 13$.

`mulmod(5,6,13)$SingleInteger`

$$4$$

<div align="right">Type: SingleInteger</div>

To reduce a small integer modulo a prime, use **positiveRemainder**.

`positiveRemainder(37,13)$SingleInteger`

$$11$$

<div align="right">Type: SingleInteger</div>

Operations **And**, **Or**, **xor**, and **Not** provide bit level operations on small integers.

`And(3,4)$SingleInteger`

$$0$$

<div align="right">Type: SingleInteger</div>

Use `shift(int,numToShift)` to shift bits, where `i` is shifted left if `numToShift` is positive, right if negative.

`shift(1,4)$SingleInteger`

$$16$$

> Type: SingleInteger

`shift(31,-1)$SingleInteger`

$$15$$

> Type: SingleInteger

Many other operations are available for small integers, including many of those provided for `Integer`. To see the other operations, use the Browse HyperDoc facility (**??** on page **??** in Section **??** on page **??**).

9.73   SparseTable

The `SparseTable` domain provides a general purpose table type with default entries.

Here we create a table to save strings under integer keys. The value `"Try again!"` is returned if no other value has been stored for a key.

`t:  SparseTable(Integer, String, "Try again!") := table()`

$$table()$$

> Type: SparseTable(Integer,String,Try again!)

Entries can be stored in the table.

`t.3 := "Number three"`

$$\text{"Number three"}$$

> Type: String

```
t.4 := "Number four"
```

$$\text{"Number four"}$$

$$\text{Type: String}$$

These values can be retrieved as usual, but if a look up fails the default entry will be returned.

```
t.3
```

$$\text{"Number three"}$$

$$\text{Type: String}$$

```
t.2
```

$$\text{"Try again!"}$$

$$\text{Type: String}$$

To see which values are explicitly stored, the **keys** and **entries** functions can be used.

```
keys t
```

$$[4, 3]$$

$$\text{Type: List Integer}$$

```
entries t
```

$$\left[\text{"Number four"}, \text{"Number three"}\right]$$

$$\text{Type: List String}$$

If a specific table representation is required, the `GeneralSparseTable` constructor should be used. The domain `SparseTable(K, E, dflt)` is equivalent to `GeneralSparseTable(K,E,Table(K,E), dflt)`. For more information, see 9.80 on page 816 and 9.30 on page 580.

9.74   SquareMatrix

The top level matrix type in Axiom is `Matrix` (see 9.52 on page 697), which provides basic arithmetic and linear algebra functions. However, since the matrices can be of any size it is not true that any pair can be added or multiplied. Thus `Matrix` has little algebraic structure.

Sometimes you want to use matrices as coefficients for polynomials or in other algebraic contexts. In this case, `SquareMatrix` should be used. The domain `SquareMatrix(n,R)` gives the ring of `n` by `n` square matrices over `R`.

Since `SquareMatrix` is not normally exposed at the top level, you must expose it before it can be used.

```
)set expose add constructor SquareMatrix
```

```
   SquareMatrix is now explicitly exposed in frame G82322
```

Once `SQMATRIX` has been exposed, values can be created using the **squareMatrix** function.

```
m := squareMatrix [ [1,-%i],[%i,4] ]
```

$$\begin{bmatrix} 1 & -i \\ i & 4 \end{bmatrix}$$

<div align="right">Type: SquareMatrix(2,Complex Integer)</div>

The usual arithmetic operations are available.

```
m*m - m
```

$$\begin{bmatrix} 1 & -4\,i \\ 4\,i & 13 \end{bmatrix}$$

<div align="right">Type: SquareMatrix(2,Complex Integer)</div>

Square matrices can be used where ring elements are required. For example, here is a matrix with matrix entries.

```
mm := squareMatrix [ [m, 1], [1-m, m**2] ]
```

$$\begin{bmatrix} \begin{bmatrix} 1 & -i \\ i & 4 \end{bmatrix} & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ \begin{bmatrix} 0 & i \\ -i & -3 \end{bmatrix} & \begin{bmatrix} 2 & -5\,i \\ 5\,i & 17 \end{bmatrix} \end{bmatrix}$$

                    Type: SquareMatrix(2,SquareMatrix(2,Complex Integer))

Or you can construct a polynomial with square matrix coefficients.

```
p := (x + m)**2
```

$$x^2 + \begin{bmatrix} 2 & -2\ i \\ 2\ i & 8 \end{bmatrix} x + \begin{bmatrix} 2 & -5\ i \\ 5\ i & 17 \end{bmatrix}$$

                    Type: Polynomial SquareMatrix(2,Complex Integer)

This value can be converted to a square matrix with polynomial coefficients.

```
p::SquareMatrix(2, ?)
```

$$\begin{bmatrix} x^2 + 2\ x + 2 & -2\ i\ x - 5\ i \\ 2\ i\ x + 5\ i & x^2 + 8\ x + 17 \end{bmatrix}$$

                    Type: SquareMatrix(2,Polynomial Complex Integer)

For more information on related topics, see **??** on page **??** in Section **??** on page **??**, **??** on page **??** in Section **??** on page **??**, and 9.52 on page 697.

9.75   SquareFreeRegularTriangularSet

The `SquareFreeRegularTriangularSet` domain constructor implements square-free regular triangular sets. See the `RegularTriangularSet` domain constructor for general regular triangular sets. Let T be a regular triangular set consisting of polynomials *t1, ..., tm* ordered by increasing main variables. The regular triangular set T is square-free if T is empty or if *t1, ..., tm-1* is square-free and if the polynomial *tm* is square-free as a univariate polynomial with coefficients in the tower of simple extensions associated with *t1, ..., tm-1*.

The main interest of square-free regular triangular sets is that their associated towers of simple extensions are product of fields. Consequently, the saturated ideal of a square-free regular triangular set is radical. This property simplifies some of the operations related to regular triangular sets. However, building square-free regular triangular sets is generally more expensive than building general regular triangular sets.

As   the   `RegularTriangularSet`   domain   constructor,   the `SquareFreeRegularTriangularSet` domain constructor also implements a method for solving polynomial systems by means of regular triangular sets. This is in fact the same method with some adaptations to take into account

the fact that the computed regular chains are square-free. Note that it is also possible to pass from a decomposition into general regular triangular sets to a decomposition into square-free regular triangular sets. This conversion is used internally by the `LazardSetSolvingPackage` package constructor.

N.B. When solving polynomial systems with the `SquareFreeRegularTriangularSet` domain constructor or the `LazardSetSolvingPackage` package constructor, decompositions have no redundant components. See also `LexTriangularPackage` and `ZeroDimensionalSolvePackage` for the case of algebraic systems with a finite number of (complex) solutions.

We shall explain now how to use the constructor `SquareFreeRegularTriangularSet`.

This constructor takes four arguments. The first one, **R**, is the coefficient ring of the polynomials; it must belong to the category `GcdDomain`. The second one, **E**, is the exponent monoid of the polynomials; it must belong to the category `OrderedAbelianMonoidSup`. the third one, **V**, is the ordered set of variables; it must belong to the category `OrderedSet`. The last one is the polynomial ring; it must belong to the category `RecursivePolynomialCategory(R,E,V)`. The abbreviation for `SquareFreeRegularTriangularSet` is `SREGSET`.

Note that the way of understanding triangular decompositions is detailed in the example of the `RegularTriangularSet` constructor.

Let us illustrate the use of this constructor with one example (Donati-Traverso). Define the coefficient ring.

```
R := Integer
```

$$Integer$$

                                          Type: Domain

Define the list of variables,

```
ls :  List Symbol := [x,y,z,t]
```

$$[x, y, z, t]$$

                                          Type: List Symbol

and make it an ordered set;

```
V := OVAR(ls)
```

$$OrderedVariableList\ [x,y,z,t]$$

<div align="right">Type: Domain</div>

then define the exponent monoid.

```
E := IndexedExponents V
```

$$\text{IndexedExponents OrderedVariableList } [x,y,z,t]$$

<div align="right">Type: Domain</div>

Define the polynomial ring.

```
P := NSMP(R, V)
```

$$\text{NewSparseMultivariatePolynomial(Integer,OrderedVariableList } [x,y,z,t])$$

<div align="right">Type: Domain</div>

Let the variables be polynomial.

```
x:  P := 'x
```

$$x$$

<div align="right">Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[x,y,z,t])</div>

```
y:  P := 'y
```

$$y$$

<div align="right">Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[x,y,z,t])</div>

```
z:  P := 'z
```

$$z$$

<div align="right">Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[x,y,z,t])</div>

```
t:  P := 't
```

$$t$$

<div align="right">Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[x,y,z,t])</div>

Now call the `SquareFreeRegularTriangularSet` domain constructor.

```
ST := SREGSET(R,E,V,P)
```

SquareFreeRegularTriangularSet(Integer,
  IndexedExponentsOrderedVariableList[x, y, z, t],
  OrderedVariableList[x, y, z, t],
  NewSparseMultivariatePolynomial(Integer,
  OrderedVariableList[x, y, z, t]))

<div align="right">Type: Domain</div>

Define a polynomial system.

```
p1 := x ** 31 - x ** 6 - x - y
```

$$x^{31} - x^6 - x - y$$

<div align="right">Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[x,y,z,t])</div>

```
p2 := x ** 8 - z
```

$$x^8 - z$$

<div align="right">Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[x,y,z,t])</div>

```
p3 := x ** 10 - t
```

$$x^{10} - t$$

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[x,y,z,t])

lp := [p1, p2, p3]

$$\left[x^{31} - x^6 - x - y, x^8 - z, x^{10} - t \right]$$

Type: List
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[x,y,z,t])

First of all, let us solve this system in the sense of Kalkbrener.

zeroSetSplit(lp)$ST

$$\left[\left\{ z^5 - t^4, t\ z\ y^2 + 2\ z^3\ y - t^8 + 2\ t^5 + t^3 - t^2, \left(t^4 - t \right)\ x - t\ y - z^2 \right\} \right]$$

Type: List SquareFreeRegularTriangularSet(Integer,
IndexedExponents OrderedVariableList [x,y,z,t],
OrderedVariableList [x,y,z,t],
NewSparseMultivariatePolynomial(Integer, OrderedVariableList
[x,y,z,t]))

And now in the sense of Lazard (or Wu and other authors).

zeroSetSplit(lp,false)$ST

$$\left[\left\{ z^5 - t^4, t\ z\ y^2 + 2\ z^3\ y - t^8 + 2\ t^5 + t^3 - t^2, \left(t^4 - t \right)\ x - t\ y - z^2 \right\}, \right.$$
$$\left. \left\{ t^3 - 1, z^5 - t, t\ y + z^2, z\ x^2 - t \right\}, \{ t, z, y, x \} \right]$$

Type: List SquareFreeRegularTriangularSet(Integer,
IndexedExponents OrderedVariableList [x,y,z,t],
OrderedVariableList [x,y,z,t],
NewSparseMultivariatePolynomial(Integer, OrderedVariableList
[x,y,z,t]))

Now to see the difference with the `RegularTriangularSet` domain constructor,
we define:

T := REGSET(R,E,V,P)

> RegularTriangularSet(Integer,
> IndexedExponentsOrderedVariableList[x, y, z, t],
> OrderedVariableList[x, y, z, t],
> NewSparseMultivariatePolynomial(Integer,
> OrderedVariableList[x, y, z, t]))

<div align="right">Type: Domain</div>

and compute:

```
lts := zeroSetSplit(lp,false)$T
```

$$\left[\left\{z^5 - t^4, t\ z\ y^2 + 2\ z^3\ y - t^8 + 2\ t^5 + t^3 - t^2, \left(t^4 - t\right)\ x - t\ y - z^2\right\},\right.$$

$$\left.\left\{t^3 - 1, z^5 - t, t\ z\ y^2 + 2\ z^3\ y + 1, z\ x^2 - t\right\}, \{t, z, y, x\}\right]$$

<div align="center">Type: List RegularTriangularSet(Integer, IndexedExponents
OrderedVariableList [x,y,z,t], OrderedVariableList [x,y,z,t],
NewSparseMultivariatePolynomial(Integer, OrderedVariableList
[x,y,z,t]))</div>

If you look at the second set in both decompositions in the sense of Lazard, you will see that the polynomial with main variable **y** is not the same.

Let us understand what has happened.

We define:

```
ts := lts.2
```

$$\left\{t^3 - 1, z^5 - t, t\ z\ y^2 + 2\ z^3\ y + 1, z\ x^2 - t\right\}$$

<div align="center">Type: RegularTriangularSet(Integer, IndexedExponents
OrderedVariableList [x,y,z,t], OrderedVariableList [x,y,z,t],
NewSparseMultivariatePolynomial(Integer, OrderedVariableList
[x,y,z,t]))</div>

```
pol := select(ts,'y)$T
```

$$t\ z\ y^2 + 2\ z^3\ y + 1$$

<div align="center">Type: Union(NewSparseMultivariatePolynomial(Integer,
OrderedVariableList [x,y,z,t]),...)</div>

```
tower := collectUnder(ts,'y)$T
```

$$\left\{ t^3 - 1, z^5 - t \right\}$$

```
             Type:  RegularTriangularSet(Integer, IndexedExponents
    OrderedVariableList [x,y,z,t], OrderedVariableList [x,y,z,t],
     NewSparseMultivariatePolynomial(Integer, OrderedVariableList
                                                   [x,y,z,t]))
```

```
pack := RegularTriangularSetGcdPackage(R,E,V,P,T)
```

> RegularTriangularSetGcdPackage(Integer,
>   IndexedExponentsOrderedVariableList[x, y, z, t],
>   OrderedVariableList[x, y, z, t],
>   NewSparseMultivariatePolynomial(Integer,
>   OrderedVariableList[x, y, z, t]),
>   RegularTriangularSet(Integer,
>   IndexedExponentsOrderedVariableList[x, y, z, t],
>   OrderedVariableList[x, y, z, t],
>   NewSparseMultivariatePolynomial(Integer,
>   OrderedVariableList[x, y, z, t])))

```
                                                 Type:   Domain
```

Then we compute:

```
toseSquareFreePart(pol,tower)$pack
```

$$\left[\left[val = t\,y + z^2, tower = \left\{ t^3 - 1, z^5 - t \right\} \right]\right]$$

```
 Type:  List Record(val:  NewSparseMultivariatePolynomial(Integer,
                          OrderedVariableList [x,y,z,t]), tower:
                   RegularTriangularSet(Integer, IndexedExponents
     OrderedVariableList [x,y,z,t], OrderedVariableList [x,y,z,t],
      NewSparseMultivariatePolynomial(Integer, OrderedVariableList
                                                   [x,y,z,t])))
```

9.76   Stream

A `Stream` object is represented as a list whose last element contains the where-withal to create the next element, should it ever be required.

Let `ints` be the infinite stream of non-negative integers.

```
ints := [i for i in 0..]
```

$$[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \ldots]$$

<div align="right">Type: Stream NonNegativeInteger</div>

By default, ten stream elements are calculated. This number may be changed to something else by the system command)set streams calculate. For the display purposes of this book, we have chosen a smaller value.

More generally, you can construct a stream by specifying its initial value and a function which, when given an element, creates the next element.

```
f :  List INT -> List INT
```

<div align="right">Void</div>

```
f x == [x.1 + x.2, x.1]
```

<div align="right">Void</div>

```
fibs := [i.2 for i in [generate(f,[1,1])]]
```

```
   Compiling function f with type List Integer -> List Integer
```

$$[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots]$$

<div align="right">Type: Stream Integer</div>

You can create the stream of odd non-negative integers by either filtering them from the integers, or by evaluating an expression for each integer.

```
[i for i in ints | odd?  i]
```

$$[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, \ldots]$$

<div align="right">Type: Stream NonNegativeInteger</div>

```
odds := [2*i+1 for i in ints]
```

$$[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, \ldots]$$

<div align="right">Type: Stream NonNegativeInteger</div>

You can accumulate the initial segments of a stream using the **scan** operation.

```
scan(0,+,odds)
```

$$[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, \ldots]$$

<div align="right">Type: Stream NonNegativeInteger</div>

The corresponding elements of two or more streams can be combined in this way.

```
[i*j for i in ints for j in odds]
```

$$[0, 3, 10, 21, 36, 55, 78, 105, 136, 171, \ldots]$$

<div align="right">Type: Stream NonNegativeInteger</div>

```
map(*,ints,odds)
```

$$[0, 3, 10, 21, 36, 55, 78, 105, 136, 171, \ldots]$$

<div align="right">Type: Stream NonNegativeInteger</div>

Many operations similar to those applicable to lists are available for streams.

```
first ints
```

<div align="center">0</div>

<div align="right">Type: NonNegativeInteger</div>

```
rest ints
```

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \ldots]$$

<div align="right">Type: Stream NonNegativeInteger</div>

```
fibs 20
```

<div align="center">

6765

</div>

<div align="right">

Type: PositiveInteger

</div>

The packages `StreamFunctions1`, `StreamFunctions2` and `StreamFunctions3` export some useful stream manipulation operations. For more information, see **??** on page **??** in Section **??** on page **??**, **??** on page **??** in Section **??** on page **??**, 9.12 on page 504, and 9.47 on page 675.

9.77   String

The type `String` provides character strings. Character strings provide all the operations for a one-dimensional array of characters, plus additional operations for manipulating text. For more information on related topics, see 9.8 on page 488 and 9.9 on page 491. You can also issue the system command `)show String` to display the full list of operations defined by `String`.

String values can be created using double quotes.

```
hello := "Hello, I'm AXIOM!"
```

<div align="center">

"Hello, I'm AXIOM!"

</div>

<div align="right">

Type: String

</div>

Note, however, that double quotes and underscores must be preceded by an extra underscore.

```
said := "Jane said, _"Look!_""
```

<div align="center">

"Jane said, "Look!""

</div>

<div align="right">

Type: String

</div>

```
saw := "She saw exactly one underscore:  __."
```

<div align="center">

"She saw exactly one underscore: _."

</div>

<div align="right">

Type: String

</div>

It is also possible to use **new** to create a string of any size filled with a given character. Since there are many **new** functions it is necessary to indicate the desired type.

```
gasp:  String := new(32, char "x")
```

```
"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
```

                                                     Type: String

The length of a string is given by "**#**".

```
#gasp
```

                                    32

                                                     Type: PositiveInteger

Indexing operations allow characters to be extracted or replaced in strings. For any string `s`, indices lie in the range `1..#s`.

```
hello.2
```

                                     e

                                                     Type: Character

Indexing is really just the application of a string to a subscript, so any application syntax works.

```
hello 2
```

                                     e

                                                     Type: Character

```
hello(2)
```

                                     e

                                                     Type: Character

If it is important not to modify a given string, it should be copied before any updating operations are used.

```
hullo := copy hello
```

$$\text{"Hello, I'm AXIOM!"}$$

<div align="right">Type: String</div>

```
hullo.2 := char "u"; [hello, hullo]
```

$$\left[\text{"Hello, I'm AXIOM!"},\text{"Hullo, I'm AXIOM!"}\right]$$

<div align="right">Type: List String</div>

Operations are provided to split and join strings. The **concat** operation allows several strings to be joined together.

```
saidsaw := concat ["alpha","---","omega"]
```

$$\text{"alpha---omega"}$$

<div align="right">Type: String</div>

There is a version of **concat** that works with two strings.

```
concat("hello ","goodbye")
```

$$\text{"hello goodbye"}$$

<div align="right">Type: String</div>

Juxtaposition can also be used to concatenate strings.

```
"This " "is " "several " "strings " "concatenated."
```

$$\text{"This is several strings concatenated."}$$

<div align="right">Type: String</div>

Substrings are obtained by giving an index range.

```
hello(1..5)
```

$$\text{"Hello"}$$

<div align="right">Type: String</div>

```
hello(8..)
```

$$\text{"I'm AXIOM!"}$$

<div align="right">Type: String</div>

A string can be split into several substrings by giving a separation character or character class.

```
split(hello, char " ")
```

$$\left[\texttt{"Hello,","I'm","AXIOM!"}\right]$$

<div align="right">Type: List String</div>

```
other := complement alphanumeric();
```

<div align="right">Type: CharacterClass</div>

```
split(saidsaw, other)
```

$$\left[\texttt{"alpha","omega"}\right]$$

<div align="right">Type: List String</div>

Unwanted characters can be trimmed from the beginning or end of a string using the operations **trim**, **leftTrim** and **rightTrim**.

```
trim("## ++ relax ++ ##", char "#")
```

$$\texttt{" ++ relax ++ "}$$

<div align="right">Type: String</div>

Each of these functions takes a string and a second argument to specify the characters to be discarded.

```
trim("## ++ relax ++ ##", other)
```

                                "relax"

                                                        Type: String

The second argument can be given either as a single character or as a character class.

```
leftTrim ("## ++ relax ++ ##", other)
```

                                "relax ++ ##"

                                                        Type: String

```
rightTrim("## ++ relax ++ ##", other)
```

                                "## ++ relax"

                                                        Type: String

Strings can be changed to upper case or lower case using the operations **upperCase**, **upperCase**, **lowerCase** and **lowerCase**.

```
upperCase hello
```

                        "HELLO, I'M AXIOM!"

                                                        Type: String

The versions with the exclamation mark change the original string, while the others produce a copy.

```
lowerCase hello
```

                        "hello, i'm axiom!"

                                                        Type: String

Some basic string matching is provided. The function **prefix?** tests whether
one string is an initial prefix of another.

```
prefix?("He", "Hello")
```

<div align="center">true</div>

<div align="right">Type: Boolean</div>

```
prefix?("Her", "Hello")
```

<div align="center">false</div>

<div align="right">Type: Boolean</div>

A similar function, **suffix?**, tests for suffixes.

```
suffix?("", "Hello")
```

<div align="center">true</div>

<div align="right">Type: Boolean</div>

```
suffix?("LO", "Hello")
```

<div align="center">false</div>

<div align="right">Type: Boolean</div>

The function **substring?** tests for a substring given a starting position.

```
substring?("ll", "Hello", 3)
```

<div align="center">true</div>

<div align="right">Type: Boolean</div>

```
substring?("ll", "Hello", 4)
```

<div align="center">false</div>

                                                    Type: Boolean

A number of **position** functions locate things in strings. If the first argument to
position is a string, then `position(s,t,i)` finds the location of `s` as a substring
of `t` starting the search at position `i`.

```
n := position("nd", "underground", 1)
```

                                 2

                                        Type: PositiveInteger

```
n := position("nd", "underground", n+1)
```

                                10

                                        Type: PositiveInteger

If `s` is not found, then `0` is returned (`minIndex(s)-1` in `IndexedString`).

```
n := position("nd", "underground", n+1)
```

                                 0

                                        Type: NonNegativeInteger

To search for a specific character or a member of a character class, a different
first argument is used.

```
position(char "d", "underground", 1)
```

                                 3

                                        Type: PositiveInteger

```
position(hexDigit(), "underground", 1)
```

                                 3

                                        Type: PositiveInteger

9.78   StringTable

This domain provides a table type in which the keys are known to be strings so special techniques can be used. Other than performance, the type `StringTable(S)` should behave exactly the same way as `Table(String,S)`. See 9.80 on page 816 for general information about tables.

This creates a new table whose keys are strings.

```
t:  StringTable(Integer) := table()
```

$$table()$$

```
                           Type:  StringTable Integer
```

The value associated with each string key is the number of characters in the string.

```
for s in split("My name is Ian Watt.",char " ")
  repeat
    t.s := #s
```

```
                                                    Void
```

```
for key in keys t repeat output [key, t.key]
```

```
    ["Ian",3]
    ["My",2]
    ["Watt.",5]
    ["name",4]
    ["is",2]
```

```
                                                    Void
```

9.79   Symbol

Symbols are one of the basic types manipulated by Axiom. The `Symbol` domain provides ways to create symbols of many varieties.

The simplest way to create a symbol is to "single quote" an identifier.

```
X: Symbol := 'x
```

$$x$$

<div align="right">Type: Symbol</div>

This gives the symbol even if `x` has been assigned a value. If `x` has not been assigned a value, then it is possible to omit the quote.

`XX: Symbol := x`

$$x$$

<div align="right">Type: Symbol</div>

Declarations must be used when working with symbols, because otherwise the interpreter tries to place values in a more specialized type `Variable`.

`A := 'a`

$$a$$

<div align="right">Type: Variable a</div>

`B := b`

$$b$$

<div align="right">Type: Variable b</div>

The normal way of entering polynomials uses this fact.

`x**2 + 1`

$$x^2 + 1$$

<div align="right">Type: Polynomial Integer</div>

Another convenient way to create symbols is to convert a string. This is useful when the name is to be constructed by a program.

`"Hello"::Symbol`

$$Hello$$

<div align="right">Type: Symbol</div>

Sometimes it is necessary to generate new unique symbols, for example, to name constants of integration. The expression `new()` generates a symbol starting with `%`.

`new()$Symbol`

$$\%A$$

<div align="right">Type: Symbol</div>

Successive calls to **new** produce different symbols.

`new()$Symbol`

$$\%B$$

<div align="right">Type: Symbol</div>

The expression `new("s")` produces a symbol starting with `%s`.

`new("xyz")$Symbol`

$$\%xyz0$$

<div align="right">Type: Symbol</div>

A symbol can be adorned in various ways. The most basic thing is applying a symbol to a list of subscripts.

`X[i,j]`

$$x_{i,j}$$

<div align="right">Type: Symbol</div>

Somewhat less pretty is to attach subscripts, superscripts or arguments.

`U := subscript(u, [1,2,1,2])`

$$u_{1,2,1,2}$$

<div align="right">Type: Symbol</div>

```
V := superscript(v, [n])
```

$$v^n$$

<div align="right">Type: Symbol</div>

```
P := argscript(p, [t])
```

$$p\,(t)$$

<div align="right">Type: Symbol</div>

It is possible to test whether a symbol has scripts using the **scripted?** test.

```
scripted?  U
```

<div align="center">true</div>

<div align="right">Type: Boolean</div>

```
scripted?  X
```

<div align="center">false</div>

<div align="right">Type: Boolean</div>

If a symbol is not scripted, then it may be converted to a string.

```
string X
```

<div align="center">"x"</div>

<div align="right">Type: String</div>

The basic parts can always be extracted using the **name** and **scripts** operations.

```
name U
```

$$u$$

<div align="right">Type: Symbol</div>

scripts U

$$[sub = [1, 2, 1, 2], sup = [], presup = [], presub = [], args = []]$$

Type: Record(sub: List OutputForm, sup: List OutputForm,
presup: List OutputForm, presub: List OutputForm, args: List
OutputForm)

name X

$$x$$

<div align="right">Type: Symbol</div>

scripts X

$$[sub = [], sup = [], presup = [], presub = [], args = []]$$

Type: Record(sub: List OutputForm, sup: List OutputForm,
presup: List OutputForm, presub: List OutputForm, args: List
OutputForm)

The most general form is obtained using the **script** operation. This operation
takes an argument which is a list containing, in this order, lists of subscripts,
superscripts, presuperscripts, presubscripts and arguments to a symbol.

```
M := script(Mammoth, [ [i,j],[k,l],[0,1],[2],[u,v,w] ])
```

$$\substack{0,1 \\ 2} Mammoth_{i,j}^{k,l} (u, v, w)$$

<div align="right">Type: Symbol</div>

scripts M

$$[sub = [i, j], sup = [k, l], presup = [0, 1], presub = [2], args = [u, v, w]]$$

```
      Type:  Record( sub:  List OutputForm, sup:  List OutputForm,
   presup:  List OutputForm, presub:  List OutputForm, args:  List
                                                         OutputForm)
```

If trailing lists of scripts are omitted, they are assumed to be empty.

```
N := script(Nut, [ [i,j],[k,l],[0,1] ])
```

$$^{0,1}Nut_{i,j}^{k,l}$$

```
                                                     Type:  Symbol
```

```
scripts N
```

$$[sub = [i,j], sup = [k,l], presup = [0,1], presub = [], args = []]$$

```
      Type:  Record( sub:  List OutputForm, sup:  List OutputForm,
   presup:  List OutputForm, presub:  List OutputForm, args:  List
                                                         OutputForm)
```

9.80   Table

The `Table` constructor provides a general structure for associative storage. This type provides hash tables in which data objects can be saved according to keys of any type. For a given table, specific types must be chosen for the keys and entries.

In this example the keys to the table are polynomials with integer coefficients. The entries in the table are strings.

```
t:  Table(Polynomial Integer, String) := table()
```

$$table()$$

```
                           Type:  Table(Polynomial Integer,String)
```

To save an entry in the table, the **setelt** operation is used. This can be called directly, giving the table a key and an entry.

```
setelt(t, x**2 - 1, "Easy to factor")
```

```
                           "Easy to factor"
```

                                                    Type: String

Alternatively, you can use assignment syntax.

```
t(x**3 + 1) := "Harder to factor"
```

                          "Harder to factor"

                                                    Type: String

```
t(x) := "The easiest to factor"
```

                          "The easiest to factor"

                                                    Type: String

Entries are retrieved from the table by calling the **elt** operation.

```
elt(t, x)
```

                          "The easiest to factor"

                                                    Type: String

This operation is called when a table is "applied" to a key using this or the
following syntax.

```
t.x
```

                          "The easiest to factor"

                                                    Type: String

```
t x
```

                          "The easiest to factor"

                                                    Type: String

Parentheses are used only for grouping. They are needed if the key is an infixed
expression.

```
t.(x**2 - 1)
```

<div align="center">"Easy to factor"</div>

<div align="right">Type: String</div>

Note that the **elt** operation is used only when the key is known to be in the table—otherwise an error is generated.

```
t (x**3 + 1)
```

<div align="center">"Harder to factor"</div>

<div align="right">Type: String</div>

You can get a list of all the keys to a table using the **keys** operation.

```
keys t
```

$$\left[x, x^3 + 1, x^2 - 1\right]$$

<div align="right">Type: List Polynomial Integer</div>

If you wish to test whether a key is in a table, the **search** operation is used. This operation returns either an entry or **"failed"**.

```
search(x, t)
```

<div align="center">"The easiest to factor"</div>

<div align="right">Type: Union(String,...)</div>

```
search(x**2, t)
```

<div align="center">"failed"</div>

<div align="right">Type: Union("failed",...)</div>

The return type is a union so the success of the search can be tested using `case`.

```
search(x**2, t) case "failed"
```

<div align="center">true</div>

<div align="right">Type: Boolean</div>

The **remove** operation is used to delete values from a table.

```
remove!(x**2-1, t)
```

<div align="center">"Easy to factor"</div>

<div align="right">Type: Union(String,...)</div>

If an entry exists under the key, then it is returned. Otherwise **remove** returns
`"failed"`.

```
remove!(x-1, t)
```

<div align="center">"failed"</div>

<div align="right">Type: Union("failed",...)</div>

The number of key-entry pairs can be found using the # operation.

```
#t
```

<div align="center">2</div>

<div align="right">Type: PositiveInteger</div>

Just as **keys** returns a list of keys to the table, a list of all the entries can be
obtained using the **members** operation.

```
members t
```

<div align="center">["The easiest to factor", "Harder to factor"]</div>

<div align="right">Type: List String</div>

A number of useful operations take functions and map them on to the table to
compute the result. Here we count the entries which have "Hard" as a prefix.

```
count(s:  String +-> prefix?("Hard", s), t)
```

                                       1

                                                    Type: PositiveInteger

Other table types are provided to support various needs.

> `AssociationList` gives a list with a table view. This allows new entries to
> be appended onto the front of the list to cover up old entries. This is useful
> when table entries need to be stacked or when frequent list traversals are
> required. See 9.1 on page 459 for more information.

> `EqTable` gives tables in which keys are considered equal only when they
> are in fact the same instance of a structure. See 9.18 on page 536 for more
> information.

> `StringTable` should be used when the keys are known to be strings. See
> 9.78 on page 811 for more information.

> `SparseTable` provides tables with default entries, so lookup never fails.
> The `GeneralSparseTable` constructor can be used to make any table type
> behave this way. See 9.73 on page 792 for more information.

> `KeyedAccessFile` allows values to be saved in a file, accessed as a table.
> See 9.38 on page 608 for more information.

9.81   TextFile

The domain `TextFile` allows Axiom to read and write character data and ex-
change text with other programs. This type behaves in Axiom much like a `File`
of strings, with additional operations to cause new lines. We give an example
of how to produce an upper case copy of a file.

This is the file from which we read the text.

```
f1:  TextFile := open("/etc/group", "input")
```

                               "/etc/group"

                                                        Type: TextFile

This is the file to which we write the text.

```
f2:  TextFile := open("/tmp/MOTD", "output")
```

                               "/tmp/MOTD"

```
                                                        Type:  TextFile
```

Entire lines are handled using the **readLine** and **writeLine** operations.

```
l := readLine!  f1
```

```
                            "root:x:0:root"
```

```
                                                          Type:  String
```

```
writeLine!(f2, upperCase l)
```

```
                            "ROOT:X:0:ROOT"
```

```
                                                          Type:  String
```

Use the **endOfFile?** operation to check if you have reached the end of the file.

```
while not endOfFile? f1 repeat
    s := readLine! f1
    writeLine!(f2, upperCase s)
```

```
                                                                   Void
```

The file `f1` is exhausted and should be closed.

```
close!  f1
```

```
                            "/etc/group"
```

```
                                                        Type:  TextFile
```

It is sometimes useful to write lines a bit at a time. The **write** operation allows
this.

```
write!(f2, "-The-")
```

```
                            "-The-"
```

```
                                                          Type:  String
```

```
write!(f2, "-End-")
```

$$\text{"-End-"}$$

<div align="right">Type: String</div>

This ends the line. This is done in a machine-dependent manner.

```
writeLine!  f2
```

$$\text{""}$$

<div align="right">Type: String</div>

```
close!  f2
```

$$\text{"/tmp/MOTD"}$$

<div align="right">Type: TextFile</div>

Finally, clean up.

```
)system rm /tmp/MOTD
```

For more information on related topics, see 9.24 on page 555, 9.38 on page 608, and 9.41 on page 650.

9.82   TwoDimensionalArray

The `TwoDimensionalArray` domain is used for storing data in a two dimensional data structure indexed by row and by column. Such an array is a homogeneous data structure in that all the entries of the array must belong to the same Axiom domain (although see ?? on page ?? in Section ?? on page ??). Each array has a fixed number of rows and columns specified by the user and arrays are not extensible. In Axiom, the indexing of two-dimensional arrays is one-based. This means that both the "first" row of an array and the "first" column of an array are given the index 1. Thus, the entry in the upper left corner of an array is in position (1,1).

The operation **new** creates an array with a specified number of rows and columns and fills the components of that array with a specified entry. The arguments of this operation specify the number of rows, the number of columns, and the entry.

This creates a five-by-four array of integers, all of whose entries are zero.

```
arr :  ARRAY2 INT := new(5,4,0)
```

$$
\begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{bmatrix}
$$

<div align="right">Type: TwoDimensionalArray Integer</div>

The entries of this array can be set to other integers using the operation **setelt**.
Issue this to set the element in the upper left corner of this array to 17.

```
setelt(arr,1,1,17)
```

<div align="center">17</div>

<div align="right">Type: PositiveInteger</div>

Now the first element of the array is 17.

```
arr
```

$$
\begin{bmatrix}
17 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{bmatrix}
$$

<div align="right">Type: TwoDimensionalArray Integer</div>

Likewise, elements of an array are extracted using the operation **elt**.

```
elt(arr,1,1)
```

<div align="center">17</div>

<div align="right">Type: PositiveInteger</div>

Another way to use these two operations is as follows. This sets the element in
position (3,2) of the array to 15.

```
arr(3,2) := 15
```

$$15$$

<div align="right">Type: PositiveInteger</div>

This extracts the element in position `(3,2)` of the array.

`arr(3,2)`

$$15$$

<div align="right">Type: PositiveInteger</div>

The operations **elt** and **setelt** come equipped with an error check which verifies that the indices are in the proper ranges. For example, the above array has five rows and four columns, so if you ask for the entry in position `(6,2)` with `arr(6,2)` Axiom displays an error message. If there is no need for an error check, you can call the operations **qelt** and **qsetelt** which provide the same functionality but without the error check. Typically, these operations are called in well-tested programs.

The operations **row** and **column** extract rows and columns, respectively, and return objects of `OneDimensionalArray` with the same underlying element type.

`row(arr,1)`

$$[17, 0, 0, 0]$$

<div align="right">Type: OneDimensionalArray Integer</div>

`column(arr,1)`

$$[17, 0, 0, 0, 0]$$

<div align="right">Type: OneDimensionalArray Integer</div>

You can determine the dimensions of an array by calling the operations **nrows** and **ncols**, which return the number of rows and columns, respectively.

`nrows(arr)`

$$5$$

<div align="right">Type: PositiveInteger</div>

```
ncols(arr)
```

$$4$$

<div align="right">Type: PositiveInteger</div>

To apply an operation to every element of an array, use **map**. This creates a new array. This expression negates every element.

```
map(-,arr)
```

$$
\begin{bmatrix}
-17 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & -15 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{bmatrix}
$$

<div align="right">Type: TwoDimensionalArray Integer</div>

This creates an array where all the elements are doubled.

```
map((x +-> x + x),arr)
```

$$
\begin{bmatrix}
34 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 30 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{bmatrix}
$$

<div align="right">Type: TwoDimensionalArray Integer</div>

To change the array destructively, use **map** instead of **map**. If you need to make a copy of any array, use **copy**.

```
arrc := copy(arr)
```

$$
\begin{bmatrix}
17 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 15 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{bmatrix}
$$

<div align="right">Type: TwoDimensionalArray Integer</div>

`map!(-,arrc)`

$$\begin{bmatrix} -17 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -15 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

<div align="right">Type: TwoDimensionalArray Integer</div>

`arrc`

$$\begin{bmatrix} -17 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -15 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

<div align="right">Type: TwoDimensionalArray Integer</div>

`arr`

$$\begin{bmatrix} 17 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 15 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

<div align="right">Type: TwoDimensionalArray Integer</div>

Use **member?** to see if a given element is in an array.

`member?(17,arr)`

<div align="center">true</div>

<div align="right">Type: Boolean</div>

`member?(10317,arr)`

<div align="center">false</div>

```
                                              Type:  Boolean
```

To see how many times an element appears in an array, use **count**.

```
count(17,arr)
```

$$1$$

```
                                         Type:  PositiveInteger
```

```
count(0,arr)
```

$$18$$

```
                                         Type:  PositiveInteger
```

For more information about the operations available for `TwoDimensionalArray`, issue `)show TwoDimensionalArray`. For information on related topics, see 9.52 on page 697 and 9.57 on page 715.

9.83 UnivariatePolynomial

The domain constructor `UnivariatePolynomial` (abbreviated `UP`) creates domains of univariate polynomials in a specified variable. For example, the domain `UP(a1,POLY FRAC INT)` provides polynomials in the single variable `a1` whose coefficients are general polynomials with rational number coefficients.

Restriction:

Axiom does not allow you to create types where `UnivariatePolynomial` is contained in the coefficient type of `Polynomial`. Therefore, `UP(x,POLY INT)` is legal but `POLY UP(x,INT)` is not.

`UP(x,INT)` is the domain of polynomials in the single variable `x` with integer coefficients.

```
(p,q) :  UP(x,INT)
```

```
                                                       Void
```

```
p := (3*x-1)**2 * (2*x + 8)
```

$$18\ x^3 + 60\ x^2 - 46\ x + 8$$

<div align="right">Type: UnivariatePolynomial(x,Integer)</div>

```
q := (1 - 6*x + 9*x**2)**2
```

$$81\ x^4 - 108\ x^3 + 54\ x^2 - 12\ x + 1$$

<div align="right">Type: UnivariatePolynomial(x,Integer)</div>

The usual arithmetic operations are available for univariate polynomials.

```
p**2 + p*q
```

$$1458\ x^7 + 3240\ x^6 - 7074\ x^5 + 10584\ x^4 - 9282\ x^3 + 4120\ x^2 - 878\ x + 72$$

<div align="right">Type: UnivariatePolynomial(x,Integer)</div>

The operation **leadingCoefficient** extracts the coefficient of the term of highest degree.

```
leadingCoefficient p
```

$$18$$

<div align="right">Type: PositiveInteger</div>

The operation **degree** returns the degree of the polynomial. Since the polynomial has only one variable, the variable is not supplied to operations like **degree**.

```
degree p
```

$$3$$

<div align="right">Type: PositiveInteger</div>

The reductum of the polynomial, the polynomial obtained by subtracting the term of highest order, is returned by **reductum**.

```
reductum p
```

$$60 \ x^2 - 46 \ x + 8$$

<div align="center">Type: UnivariatePolynomial(x,Integer)</div>

The operation **gcd** computes the greatest common divisor of two polynomials.

```
gcd(p,q)
```

$$9 \ x^2 - 6 \ x + 1$$

<div align="center">Type: UnivariatePolynomial(x,Integer)</div>

The operation **lcm** computes the least common multiple.

```
lcm(p,q)
```

$$162 \ x^5 + 432 \ x^4 - 756 \ x^3 + 408 \ x^2 - 94 \ x + 8$$

<div align="center">Type: UnivariatePolynomial(x,Integer)</div>

The operation **resultant** computes the resultant of two univariate polynomials. In the case of p and q, the resultant is 0 because they share a common root.

```
resultant(p,q)
```

$$0$$

<div align="center">Type: NonNegativeInteger</div>

To compute the derivative of a univariate polynomial with respect to its variable, use **D**.

```
D p
```

$$54 \ x^2 + 120 \ x - 46$$

<div align="center">Type: UnivariatePolynomial(x,Integer)</div>

Univariate polynomials can also be used as if they were functions. To evaluate a univariate polynomial at some point, apply the polynomial to the point.

```
p(2)
```

$$300$$

The same syntax is used for composing two univariate polynomials, i.e. substituting one polynomial for the variable in another. This substitutes `q` for the variable in `p`.

```
p(q)
```

$$9565938 \ x^{12} - 38263752 \ x^{11} + 70150212 \ x^{10} - 77944680 \ x^9 + 58852170 \ x^8 -$$

$$32227632 \ x^7 + 13349448 \ x^6 - 4280688 \ x^5 + 1058184 \ x^4 -$$

$$192672 \ x^3 + 23328 \ x^2 - 1536 \ x + 40$$

This substitutes `p` for the variable in `q`.

```
q(p)
```

$$8503056 \ x^{12} + 113374080 \ x^{11} + 479950272 \ x^{10} + 404997408 \ x^9 -$$

$$1369516896 \ x^8 - 626146848 \ x^7 + 2939858712 \ x^6 - 2780728704 \ x^5 +$$

$$1364312160 \ x^4 - 396838872 \ x^3 + 69205896 \ x^2 - 6716184 \ x + 279841$$

To obtain a list of coefficients of the polynomial, use **coefficients**.

```
l := coefficients p
```

$$[18, 60, -46, 8]$$

From this you can use **gcd** and **reduce** to compute the content of the polynomial.

```
reduce(gcd,l)
```

$$2$$

<div align="right">Type: PositiveInteger</div>

Alternatively (and more easily), you can just call **content**.

```
content p
```

$$2$$

<div align="right">Type: PositiveInteger</div>

Note that the operation **coefficients** omits the zero coefficients from the list. Sometimes it is useful to convert a univariate polynomial to a vector whose i-th position contains the degree `i-1` coefficient of the polynomial.

```
ux := (x**4+2*x+3)::UP(x,INT)
```

$$x^4 + 2\ x + 3$$

<div align="right">Type: UnivariatePolynomial(x,Integer)</div>

To get a complete vector of coefficients, use the operation **vectorise**, which takes a univariate polynomial and an integer denoting the length of the desired vector.

```
vectorise(ux,5)
```

$$[3, 2, 0, 0, 1]$$

<div align="right">Type: Vector Integer</div>

It is common to want to do something to every term of a polynomial, creating a new polynomial in the process.

This is a function for iterating across the terms of a polynomial, squaring each term.

```
squareTerms(p) == reduce(+,[t**2 for t in monomials p])
```

<div align="right">Void</div>

Recall what `p` looked like.

`p`

$$18\ x^3 + 60\ x^2 - 46\ x + 8$$

<div align="center">Type: UnivariatePolynomial(x,Integer)</div>

We can demonstrate `squareTerms` on `p`.

`squareTerms p`

```
Compiling function squareTerms with type
  UnivariatePolynomial(x,Integer) ->
    UnivariatePolynomial(x,Integer)
```

$$324\ x^6 + 3600\ x^4 + 2116\ x^2 + 64$$

<div align="center">Type: UnivariatePolynomial(x,Integer)</div>

When the coefficients of the univariate polynomial belong to a field,[7] it is possible to compute quotients and remainders.

`(r,s)  :  UP(a1,FRAC INT)`

<div align="right">Void</div>

`r := a1**2 - 2/3`

$$a1^2 - \frac{2}{3}$$

<div align="center">Type: UnivariatePolynomial(a1,Fraction Integer)</div>

`s := a1 + 4`

$$a1 + 4$$

---

[7]For example, when the coefficients are rational numbers, as opposed to integers. The important property of a field is that non-zero elements can be divided and produce another element. The quotient of the integers 2 and 3 is not another integer.

When the coefficients are rational numbers or rational expressions, the operation **quo** computes the quotient of two polynomials.

`r quo s`

$$a1 - 4$$

Type: UnivariatePolynomial(a1,Fraction Integer)

The operation **rem** computes the remainder.

`r rem s`

$$\frac{46}{3}$$

Type: UnivariatePolynomial(a1,Fraction Integer)

The operation **divide** can be used to return a record of both components.

`d := divide(r, s)`

$$\left[quotient = a1 - 4, remainder = \frac{46}{3} \right]$$

Type: Record(quotient: UnivariatePolynomial(a1,Fraction
Integer), remainder: UnivariatePolynomial(a1,Fraction Integer))

Now we check the arithmetic!

`r - (d.quotient * s + d.remainder)`

$$0$$

Type: UnivariatePolynomial(a1,Fraction Integer)

It is also possible to integrate univariate polynomials when the coefficients belong to a field.

`integrate r`

$$\frac{1}{3}\ a1^3 - \frac{2}{3}\ a1$$

<div align="center">Type: UnivariatePolynomial(a1,Fraction Integer)</div>

```
integrate s
```

$$\frac{1}{2}\ a1^2 + 4\ a1$$

<div align="center">Type: UnivariatePolynomial(a1,Fraction Integer)</div>

One application of univariate polynomials is to see expressions in terms of a specific variable.

We start with a polynomial in **a1** whose coefficients are quotients of polynomials in **b1** and **b2**.

```
t :  UP(a1,FRAC POLY INT)
```

<div align="right">Void</div>

Since in this case we are not talking about using multivariate polynomials in only two variables, we use **Polynomial**. We also use **Fraction** because we want fractions.

```
t := a1**2 - a1/b2 + (b1**2-b1)/(b2+3)
```

$$a1^2 - \frac{1}{b2}\ a1 + \frac{b1^2 - b1}{b2 + 3}$$

<div align="center">Type: UnivariatePolynomial(a1,Fraction Polynomial Integer)</div>

We push all the variables into a single quotient of polynomials.

```
u :  FRAC POLY INT := t
```

$$\frac{a1^2\ b2^2 + \left(b1^2 - b1 + 3\ a1^2 - a1\right)\ b2 - 3\ a1}{b2^2 + 3\ b2}$$

<div align="center">Type: Fraction Polynomial Integer</div>

Alternatively, we can view this as a polynomial in the variable This is a *mode-directed* conversion: you indicate as much of the structure as you care about and let Axiom decide on the full type and how to do the transformation.

```
u :: UP(b1,?)
```

$$\frac{1}{b2+3}\ b1^2 - \frac{1}{b2+3}\ b1 + \frac{a1^2\ b2 - a1}{b2}$$

```
        Type:  UnivariatePolynomial(b1,Fraction Polynomial Integer)
```

See **??** on page **??** in Section **??** on page **??** for a discussion of the factorization facilities in Axiom for univariate polynomials. For more information on related topics, see **??** on page **??** in Section **??** on page **??**, **??** on page **??** in Section **??** on page **??**, 9.63 on page 734, 9.54 on page 709, and 9.16 on page 531.

9.84 UniversalSegment

The `UniversalSegment` domain generalizes `Segment` by allowing segments without a "hi" end point.

```
pints := 1..
```

$$1..$$

```
                Type:  UniversalSegment PositiveInteger
```

```
nevens := (0..)  by -2
```

$$0..by\ -2$$

```
            Type:  UniversalSegment NonNegativeInteger
```

Values of type `Segment` are automatically converted to type `UniversalSegment` when appropriate.

```
useg:  UniversalSegment(Integer) := 3..10
```

$$3..10$$

```
                    Type:  UniversalSegment Integer
```

The operation **hasHi** is used to test whether a segment has a `hi` end point.

```
hasHi pints
```

<div align="center">false</div>

<div align="right">Type: Boolean</div>

hasHi nevens

<div align="center">false</div>

<div align="right">Type: Boolean</div>

hasHi useg

<div align="center">true</div>

<div align="right">Type: Boolean</div>

All operations available on type `Segment` apply to `UniversalSegment`, with the proviso that expansions produce streams rather than lists. This is to accommodate infinite expansions.

expand pints

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \ldots]$$

<div align="right">Type: Stream Integer</div>

expand nevens

$$[0, -2, -4, -6, -8, -10, -12, -14, -16, -18, \ldots]$$

<div align="right">Type: Stream Integer</div>

expand [1, 3, 10..15, 100..]

$$[1, 3, 10, 11, 12, 13, 14, 15, 100, 101, \ldots]$$

<div align="right">Type: Stream Integer</div>

For more information on related topics, see 9.69 on page 783, 9.70 on page 785, 9.47 on page 675, and 9.76 on page 801.

9.85 Vector

The `Vector` domain is used for storing data in a one-dimensional indexed data structure. A vector is a homogeneous data structure in that all the components of the vector must belong to the same Axiom domain. Each vector has a fixed length specified by the user; vectors are not extensible. This domain is similar to the `OneDimensionalArray` domain, except that when the components of a `Vector` belong to a `Ring`, arithmetic operations are provided. For more examples of operations that are defined for both `Vector` and `OneDimensionalArray`, see 9.57 on page 715.

As with the `OneDimensionalArray` domain, a `Vector` can be created by calling the operation **new**, its components can be accessed by calling the operations **elt** and **qelt**, and its components can be reset by calling the operations **setelt** and **qsetelt**.

This creates a vector of integers of length 5 all of whose components are 12.

```
u :  VECTOR INT := new(5,12)
```

$$[12, 12, 12, 12, 12]$$

                                        Type: Vector Integer

This is how you create a vector from a list of its components.

```
v :  VECTOR INT := vector([1,2,3,4,5])
```

$$[1, 2, 3, 4, 5]$$

                                        Type: Vector Integer

Indexing for vectors begins at 1. The last element has index equal to the length of the vector, which is computed by "**#**".

```
#(v)
```

$$5$$

                                        Type: PositiveInteger

This is the standard way to use **elt** to extract an element. Functionally, it is the same as if you had typed `elt(v,2)`.

```
v.2
```

$$2$$

<div align="right">Type: PositiveInteger</div>

This is the standard way to use **setelt** to change an element. It is the same as if you had typed `setelt(v,3,99)`.

`v.3 := 99`

$$99$$

<div align="right">Type: PositiveInteger</div>

Now look at `v` to see the change. You can use **qelt** and **qsetelt** (instead of **elt** and **setelt**, respectively) but *only* when you know that the index is within the valid range.

`v`

$$[1, 2, 99, 4, 5]$$

<div align="right">Type: Vector Integer</div>

When the components belong to a `Ring`, Axiom provides arithmetic operations for `Vector`. These include left and right scalar multiplication.

`5 * v`

$$[5, 10, 495, 20, 25]$$

<div align="right">Type: Vector Integer</div>

`v * 7`

$$[7, 14, 693, 28, 35]$$

<div align="right">Type: Vector Integer</div>

`w :  VECTOR INT := vector([2,3,4,5,6])`

$$[2, 3, 4, 5, 6]$$

<div align="right">Type: Vector Integer</div>

Addition and subtraction are also available.

```
v + w
```

$$[3, 5, 103, 9, 11]$$

<div align="right">Type: Vector Integer</div>

Of course, when adding or subtracting, the two vectors must have the same length or an error message is displayed.

```
v - w
```

$$[-1, -1, 95, -1, -1]$$

<div align="right">Type: Vector Integer</div>

For more information about other aggregate domains, see the following: 9.47 on page 675, 9.52 on page 697, 9.57 on page 715, 9.71 on page 786, 9.80 on page 816, and 9.82 on page 822. Issue the system command `)show Vector` to display the full list of operations defined by `Vector`.

9.86 Void

When an expression is not in a value context, it is given type `Void`. For example, in the expression

```
  r := (a; b; if c then d else e; f)
```

values are used only from the subexpressions `c` and `f`: all others are thrown away. The subexpressions `a`, `b`, `d` and `e` are evaluated for side-effects only and have type `Void`. There is a unique value of type `Void`.

You will most often see results of type `Void` when you declare a variable.

```
a :  Integer
```

<div align="right">Void</div>

Usually no output is displayed for `Void` results. You can force the display of a rather ugly object by issuing `)set message void on`.

```
)set message void on
```

```
b :  Fraction Integer
```

<div align="center">

`"()"`

</div>

<div align="right">

Type: Void

</div>

```
)set message void off
```

All values can be converted to type `Void`.

```
3::Void
```

<div align="right">

Void

</div>

Once a value has been converted to `Void`, it cannot be recovered.

```
% ::  PositiveInteger
```

```
Cannot convert from type Void to PositiveInteger for value "()"
```

9.87   WuWenTsunTriangularSet

The `WuWenTsunTriangularSet` domain constructor implements the characteristic set method of Wu Wen Tsun. This algorithm computes a list of triangular sets from a list of polynomials such that the algebraic variety defined by the given list of polynomials decomposes into the union of the regular-zero sets of the computed triangular sets. The constructor takes four arguments. The first one, **R**, is the coefficient ring of the polynomials; it must belong to the category `IntegralDomain`. The second one, **E**, is the exponent monoid of the polynomials; it must belong to the category `OrderedAbelianMonoidSup`. The third one, **V**, is the ordered set of variables; it must belong to the category `OrderedSet`. The last one is the polynomial ring; it must belong to the category `RecursivePolynomialCategory(R,E,V)`. The abbreviation for `WuWenTsunTriangularSet` is `WUTSET`.

Let us illustrate the facilities by an example.

Define the coefficient ring.

```
R := Integer
```

$$Integer$$

<div align="right">Type: Domain</div>

Define the list of variables,

```
ls :  List Symbol := [x,y,z,t]
```

$$[x, y, z, t]$$

<div align="right">Type: List Symbol</div>

and make it an ordered set;

```
V := OVAR(ls)
```

OrderedVariableList [x,y,z,t]

<div align="right">Type: Domain</div>

then define the exponent monoid.

```
E := IndexedExponents V
```

IndexedExponents OrderedVariableList [x,y,z,t]

<div align="right">Type: Domain</div>

Define the polynomial ring.

```
P := NSMP(R, V)
```

NewSparseMultivariatePolynomial(Integer,OrderedVariableList [x,y,z,t])

<div align="right">Type: Domain</div>

Let the variables be polynomial.

```
x:  P := 'x
```

$$x$$

<div align="right">

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[x,y,z,t])

</div>

y: P := 'y

$$y$$

<div align="right">

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[x,y,z,t])

</div>

z: P := 'z

$$z$$

<div align="right">

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[x,y,z,t])

</div>

t: P := 't

$$t$$

<div align="right">

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[x,y,z,t])

</div>

Now call the `WuWenTsunTriangularSet` domain constructor.

T := WUTSET(R,E,V,P)

WuWenTsunTriangularSet(Integer,
  IndexedExponentsOrderedVariableList[x, y, z, t],
  OrderedVariableList[x, y, z, t],
  NewSparseMultivariatePolynomial(Integer,
  OrderedVariableList[x, y, z, t]))

<div align="right">

Type: Domain

</div>

Define a polynomial system.

```
p1 := x ** 31 - x ** 6 - x - y
```

$$x^{31} - x^6 - x - y$$

<div align="right">

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[x,y,z,t])

</div>

```
p2 := x ** 8 - z
```

$$x^8 - z$$

<div align="right">

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[x,y,z,t])

</div>

```
p3 := x ** 10 - t
```

$$x^{10} - t$$

<div align="right">

Type:
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[x,y,z,t])

</div>

```
lp := [p1, p2, p3]
```

$$\left[x^{31} - x^6 - x - y, x^8 - z, x^{10} - t\right]$$

<div align="right">

Type: List
NewSparseMultivariatePolynomial(Integer,OrderedVariableList
[x,y,z,t])

</div>

Compute a characteristic set of the system.

```
characteristicSet(lp)$T
```

$$\left\{z^5 - t^4,\right.$$

$$t^4\ z^2\ y^2 + 2\ t^3\ z^4\ y + \left(-t^7 + 2\ t^4 - t\right)\ z^6 + t^6\ z,$$

$$\left.\left(t^3 - 1\right)\ z^3\ x - z^3\ y - t^3\right\}$$

    Type: Union(WuWenTsunTriangularSet(Integer, IndexedExponents
     OrderedVariableList [x,y,z,t], OrderedVariableList [x,y,z,t],
      NewSparseMultivariatePolynomial(Integer, OrderedVariableList
                                          [x,y,z,t])),...)

Solve the system.

```
zeroSetSplit(lp)$T
```

$$\left[\{t, z, y, x\}, \left\{t^3 - 1, z^5 - t^4, z^3\ y + t^3, z\ x^2 - t\right\},\right.$$

$$\left\{z^5 - t^4, t^4\ z^2\ y^2 + 2\ t^3\ z^4\ y + \left(-t^7 + 2\ t^4 - t\right)\ z^6 + t^6\ z,\right.$$

$$\left.\left(t^3 - 1\right)\ z^3\ x - z^3\ y - t^3\right\}\right]$$

    Type: List WuWenTsunTriangularSet(Integer, IndexedExponents
     OrderedVariableList [x,y,z,t], OrderedVariableList [x,y,z,t],
      NewSparseMultivariatePolynomial(Integer, OrderedVariableList
                                          [x,y,z,t]))

The `RegularTriangularSet` and `SquareFreeRegularTriangularSet` domain constructors, the `LazardSetSolvingPackage` package constructors as well as, `SquareFreeRegularTriangularSet` and `ZeroDimensionalSolvePackage` package constructors also provide operations to compute triangular decompositions of algebraic varieties. These five constructor use a special kind of characteristic sets, called regular triangular sets. These special characteristic sets have better properties than the general ones. Regular triangular sets and their related concepts are presented in the paper "On the Theories of Triangular sets" By P. Aubry, D. Lazard and M. Moreno Maza (to appear in the Journal of Symbolic Computation). The decomposition algorithm (due to the third author) available in the four above constructors provide generally better timings than the characteristic set method. In fact, the `WUTSET` constructor remains interesting for the purpose of manipulating characteristic sets whereas the other constructors are more convenient for solving polynomial systems.

Note that the way of understanding triangular decompositions is detailed in the example of the `RegularTriangularSet` constructor.

9.88   XPBWPolynomial

Initialisations

```
a:Symbol := 'a
```

$$a$$

<div align="right">Type: Symbol</div>

```
b:Symbol := 'b
```

$$b$$

<div align="right">Type: Symbol</div>

```
RN := Fraction(Integer)
```

<div align="center">Fraction Integer</div>

<div align="right">Type: Domain</div>

```
word := OrderedFreeMonoid Symbol
```

<div align="center">OrderedFreeMonoid Symbol</div>

<div align="right">Type: Domain</div>

```
lword := LyndonWord(Symbol)
```

<div align="center">LyndonWord Symbol</div>

<div align="right">Type: Domain</div>

```
base := PoincareBirkhoffWittLyndonBasis Symbol
```

<div align="center">PoincareBirkhoffWittLyndonBasis Symbol</div>

<div align="right">Type: Domain</div>

```
dpoly := XDistributedPolynomial(Symbol, RN)
```

<div align="center">XDistributedPolynomial(Symbol,Fraction Integer)</div>

<div align="right">Type: Domain</div>

```
rpoly := XRecursivePolynomial(Symbol, RN)
```

$$XRecursivePolynomial(Symbol,Fraction\ Integer)$$

<div align="right">Type: Domain</div>

```
lpoly := LiePolynomial(Symbol, RN)
```

$$LiePolynomial(Symbol,Fraction\ Integer)$$

<div align="right">Type: Domain</div>

```
poly := XPBWPolynomial(Symbol, RN)
```

$$XPBWPolynomial(Symbol,Fraction\ Integer)$$

<div align="right">Type: Domain</div>

```
liste :  List lword := LyndonWordsList([a,b], 6)
```

$$\left[[a],[b],[a\ b],\left[a^2\ b\right],\left[a\ b^2\right],\left[a^3\ b\right],\left[a^2\ b^2\right],\left[a\ b^3\right],\left[a^4\ b\right],\right.$$
$$\left[a^3\ b^2\right],\left[a^2\ b\ a\ b\right],\left[a^2\ b^3\right],\left[a\ b\ a\ b^2\right],\left[a\ b^4\right],\left[a^5\ b\right],\left[a^4\ b^2\right],$$
$$\left[a^3\ b\ a\ b\right],\left[a^3\ b^3\right],\left[a^2\ b\ a\ b^2\right],\left[a^2\ b^2\ a\ b\right],\left[a^2\ b^4\right],\left[a\ b\ a\ b^3\right],\left[a\ b^5\right]\right]$$

<div align="right">Type: List LyndonWord Symbol</div>

Let's make some polynomials

```
0$poly
```

$$0$$

<div align="right">Type: XPBWPolynomial(Symbol,Fraction Integer)</div>

```
1$poly
```

$$1$$

$$\text{Type: XPBWPolynomial(Symbol,Fraction Integer)}$$

```
p :  poly := a
```

$$[a]$$

$$\text{Type: XPBWPolynomial(Symbol,Fraction Integer)}$$

```
q :  poly := b
```

$$[b]$$

$$\text{Type: XPBWPolynomial(Symbol,Fraction Integer)}$$

```
pq:  poly := p*q
```

$$[a\ b] + [b]\ [a]$$

$$\text{Type: XPBWPolynomial(Symbol,Fraction Integer)}$$

Coerce to distributed polynomial

```
pq ::  dpoly
```

$$a\ b$$

$$\text{Type: XDistributedPolynomial(Symbol,Fraction Integer)}$$

Check some polynomial operations

```
mirror pq
```

$$[b]\ [a]$$

$$\text{Type: XPBWPolynomial(Symbol,Fraction Integer)}$$

```
ListOfTerms pq
```

$$[[k = [b]\ [a], c = 1], [k = [a\ b], c = 1]]$$

                Type: List Record(k: PoincareBirkhoffWittLyndonBasis Symbol,c:
                                                             Fraction Integer)

reductum pq

$$[a\ b]$$

                        Type: XPBWPolynomial(Symbol,Fraction Integer)

leadingMonomial pq

$$[b]\ [a]$$

                        Type: PoincareBirkhoffWittLyndonBasis Symbol

coefficients pq

$$[1, 1]$$

                                    Type: List Fraction Integer

leadingTerm pq

$$[k = [b]\ [a], c = 1]$$

          Type: Record(k: PoincareBirkhoffWittLyndonBasis Symbol,c:
                                                       Fraction Integer)

degree pq

$$2$$

                                        Type: PositiveInteger

pq4:=exp(pq,4)

$$1 + [a\ b] + [b]\ [a] + \tfrac{1}{2}\ [a\ b]\ [a\ b] + \tfrac{1}{2}\ \left[a\ b^2\right]\ [a] + \tfrac{1}{2}\ [b]\ \left[a^2\ b\right] +$$

$$\frac{3}{2}\ [b]\ [a\ b]\ [a] + \frac{1}{2}\ [b]\ [b]\ [a]\ [a]$$

```
log(pq4,4) - pq
```

$$0$$

Type: XPBWPolynomial(Symbol,Fraction Integer)

Calculations with verification in `XDistributedPolynomial`.

```
lp1 :lpoly := LiePoly liste.10
```

$$\left[a^3 \ b^2\right]$$

Type: LiePolynomial(Symbol,Fraction Integer)

```
lp2 :lpoly := LiePoly liste.11
```

$$\left[a^2 \ b \ a \ b\right]$$

Type: LiePolynomial(Symbol,Fraction Integer)

```
lp :lpoly := [lp1, lp2]
```

$$\left[a^3 \ b^2 \ a^2 \ b \ a \ b\right]$$

Type: LiePolynomial(Symbol,Fraction Integer)

```
lpd1:  dpoly := lp1
```

$$a^3 \ b^2 - 2 \ a^2 \ b \ a \ b - a^2 \ b^2 \ a + 4 \ a \ b \ a \ b \ a - a \ b^2 \ a^2 - 2 \ b \ a \ b \ a^2 + b^2 \ a^3$$

Type: XDistributedPolynomial(Symbol,Fraction Integer)

```
lpd2:  dpoly := lp2
```

$$a^2 \ b \ a \ b - a^2 \ b^2 \ a - 3 \ a \ b \ a^2 \ b + 4 \ a \ b \ a \ b \ a -$$

$$a \ b^2 \ a^2 + 2 \ b \ a^3 \ b - 3 \ b \ a^2 \ b \ a + b \ a \ b \ a^2$$

```
lpd :   dpoly := lpd1 * lpd2 - lpd2 * lpd1
```

$a^3\ b^2\ a^2\ b\ a\ b - a^3\ b^2\ a^2\ b^2\ a - 3\ a^3\ b^2\ a\ b\ a^2\ b + 4\ a^3\ b^2\ a\ b\ a\ b\ a -$
$a^3\ b^2\ a\ b^2\ a^2 + 2\ a^3\ b^3\ a^3\ b - 3\ a^3\ b^3\ a^2\ b\ a + a^3\ b^3\ a\ b\ a^2 -$
$a^2\ b\ a\ b\ a^3\ b^2 + 3\ a^2\ b\ a\ b\ a^2\ b^2\ a + 6\ a^2\ b\ a\ b\ a\ b\ a^2\ b - 12\ a^2\ b\ a\ b\ a\ b\ a\ b\ a +$
$3\ a^2\ b\ a\ b\ a\ b^2\ a^2 - 4\ a^2\ b\ a\ b^2\ a^3\ b + 6\ a^2\ b\ a\ b^2\ a^2\ b\ a - a^2\ b\ a\ b^3\ a^3 +$
$a^2\ b^2\ a^4\ b^2 - 3\ a^2\ b^2\ a^3\ b\ a\ b + 3\ a^2\ b^2\ a^2\ b\ a^2\ b - 2\ a^2\ b^2\ a\ b\ a^3\ b +$
$3\ a^2\ b^2\ a\ b\ a^2\ b\ a - 3\ a^2\ b^2\ a\ b\ a\ b\ a^2 + a^2\ b^2\ a\ b^2\ a^3 + 3\ a\ b\ a^2\ b\ a^3\ b^2 -$
$6\ a\ b\ a^2\ b\ a^2\ b\ a\ b - 3\ a\ b\ a^2\ b\ a^2\ b^2\ a + 12\ a\ b\ a^2\ b\ a\ b\ a\ b\ a - 3\ a\ b\ a^2\ b\ a\ b^2\ a^2 -$
$6\ a\ b\ a^2\ b^2\ a\ b\ a^2 + 3\ a\ b\ a^2\ b^3\ a^3 - 4\ a\ b\ a\ b\ a^4\ b^2 + 12\ a\ b\ a\ b\ a^3\ b\ a\ b -$
$12\ a\ b\ a\ b\ a^2\ b\ a^2\ b + 8\ a\ b\ a\ b\ a\ b\ a^3\ b - 12\ a\ b\ a\ b\ a\ b\ a^2\ b\ a +$
$12\ a\ b\ a\ b\ a\ b\ a\ b\ a^2 - 4\ a\ b\ a\ b\ a\ b^2\ a^3 + a\ b^2\ a^5\ b^2 - 3\ a\ b^2\ a^4\ b\ a\ b +$
$3\ a\ b^2\ a^3\ b\ a^2\ b - 2\ a\ b^2\ a^2\ b\ a^3\ b + 3\ a\ b^2\ a^2\ b\ a^2\ b\ a - 3\ a\ b^2\ a^2\ b\ a\ b\ a^2 +$
$a\ b^2\ a^2\ b^2\ a^3 - 2\ b\ a^3\ b\ a^3\ b^2 + 4\ b\ a^3\ b\ a^2\ b\ a\ b + 2\ b\ a^3\ b\ a^2\ b^2\ a -$
$8\ b\ a^3\ b\ a\ b\ a\ b\ a + 2\ b\ a^3\ b\ a\ b^2\ a^2 + 4\ b\ a^3\ b^2\ a\ b\ a^2 - 2\ b\ a^3\ b^3\ a^3 +$
$3\ b\ a^2\ b\ a^4\ b^2 - 6\ b\ a^2\ b\ a^3\ b\ a\ b - 3\ b\ a^2\ b\ a^3\ b^2\ a + 12\ b\ a^2\ b\ a^2\ b\ a\ b\ a -$
$3\ b\ a^2\ b\ a^2\ b^2\ a^2 - 6\ b\ a^2\ b\ a\ b\ a\ b\ a^2 + 3\ b\ a^2\ b\ a\ b^2\ a^3 - b\ a\ b\ a^5\ b^2 +$
$3\ b\ a\ b\ a^4\ b^2\ a + 6\ b\ a\ b\ a^3\ b\ a^2\ b - 12\ b\ a\ b\ a^3\ b\ a\ b\ a + 3\ b\ a\ b\ a^3\ b^2\ a^2 -$
$4\ b\ a\ b\ a^2\ b\ a^3\ b + 6\ b\ a\ b\ a^2\ b\ a^2\ b\ a - b\ a\ b\ a^2\ b^2\ a^3 + b^2\ a^5\ b\ a\ b -$
$b^2\ a^5\ b^2\ a - 3\ b^2\ a^4\ b\ a^2\ b + 4\ b^2\ a^4\ b\ a\ b\ a - b^2\ a^4\ b^2\ a^2 +$
$2\ b^2\ a^3\ b\ a^3\ b - 3\ b^2\ a^3\ b\ a^2\ b\ a + b^2\ a^3\ b\ a\ b\ a^2$

```
lp ::  dpoly - lpd
```

$$0$$

Calculations with verification in XRecursivePolynomial.

```
p := 3 * lp
```

$$3\ \left[a^3\ b^2\ a^2\ b\ a\ b \right]$$

Type: XPBWPolynomial(Symbol,Fraction Integer)

```
q := lp1
```

$$\left[a^3 \ b^2\right]$$

Type: XPBWPolynomial(Symbol,Fraction Integer)

pq:= p * q

$$3 \ \left[a^3 \ b^2 \ a^2 \ b \ a \ b\right] \ \left[a^3 \ b^2\right]$$

Type: XPBWPolynomial(Symbol,Fraction Integer)

pr:rpoly := p :: rpoly

$a \ (a \ (a \ b \ b \ (a \ (a \ b \ (a \ b \ 3 + b \ a \ (-3)) + b$
$(a \ (a \ b \ (-9) + b \ a \ 12) + b \ a \ a \ (-3))) + b \ a$
$(a \ (a \ b \ 6 + b \ a \ (-9)) + b \ a \ a \ 3)) + b \ (a \ b \ (a \ (a$
$(a \ b \ b \ (-3) + b \ b \ a \ 9) + b \ (a \ (a \ b \ 18 + b \ a \ (-36)) + b \ a \ a \ 9$
$)) + b \ (a \ a \ (a \ b \ (-12) + b \ a \ 18) + b \ a \ a \ a \ (-3))$
$) + b \ a \ (a \ (a \ (a \ b \ b \ 3 + b \ a \ b \ (-9)) + b \ a \ a \ b \ 9) + b \ (a$
$(a \ (a \ b \ (-6) + b \ a \ 9) + b \ a \ a \ (-9)) + b \ a \ a \ a \ 3))$
$)) + b \ (a \ (a \ b \ (a \ (a \ (a \ b \ b \ 9 + b \ (a \ b \ (-18) + b \ a$
$(-9))) + b \ (a \ b \ a \ 36 + b \ a \ a \ (-9))) + b \ (a \ b \ a \ a$
$(-18) + b \ a \ a \ a \ 9)) + b \ a \ (a \ (a \ (a \ b \ b \ (-12) + b \ a \ b \ 36) + b \ a \ a \ b$
$(-36)) + b \ (a \ (a \ (a \ b \ 24 + b \ a \ (-36)) + b \ a \ a \ 36) + b \ a \ a \ a$
$(-12)))) + b \ a \ a \ (a \ (a \ (a \ b \ b \ 3 + b \ a \ b \ (-9)$
$) + b \ a \ a \ b \ 9) + b \ (a \ (a \ (a \ b \ (-6) + b \ a \ 9) + b \ a \ a \ (-9)$
$) + b \ a \ a \ a \ 3)))) + b \ (a \ (a \ (a \ b \ (a \ (a$
$(a \ b \ b \ (-6) + b \ (a \ b \ 12 + b \ a \ 6)) + b \ (a \ b \ a \ (-24) + b \ a \ a \ 6)$
$) + b \ (a \ b \ a \ a \ 12 + b \ a \ a \ a \ (-6))) + b \ a \ (a \ (a \ (a \ b \ b \ 9 + b \ (a \ b \ (-18$
$) + b \ a \ (-9))) + b \ (a \ b \ a \ 36 + b \ a \ a \ (-9))) + b \ (a \ b \ a \ a$
$(-18) + b \ a \ a \ a \ 9))) + b \ a \ a \ (a \ (a \ (a \ b \ b \ (-3) + b \ b \ a \ 9$
$) + b \ (a \ (a \ b \ 18 + b \ a \ (-36)) + b \ a \ a \ 9)) + b \ (a \ a \ (a \ b$
$(-12) + b \ a \ 18) + b \ a \ a \ a \ (-3)))) + b \ a \ a \ a \ (a$
$(a \ b \ (a \ b \ 3 + b \ a \ (-3)) + b \ (a \ (a \ b \ (-9) + b \ a \ 12) + b \ a \ a$
$(-3))) + b \ a \ (a \ (a \ b \ 6 + b \ a \ (-9)) + b \ a \ a \ 3)))$

Type: XRecursivePolynomial(Symbol,Fraction Integer)

qr:rpoly := q :: rpoly

$a \ (a \ (a \ b \ b \ 1 + b \ (a \ b \ (-2) + b \ a \ (-1))) +$

$b \ (a \ b \ a \ 4 + b \ a \ a \ (-1))) +$

$b \ (a \ b \ a \ a \ (-2) + b \ a \ a \ a \ 1)$

Type: XRecursivePolynomial(Symbol,Fraction Integer)

```
pq ::  rpoly - pr*qr
```

$$0$$

Type: XRecursivePolynomial(Symbol,Fraction Integer)

9.89   XPolynomial

The `XPolynomial` domain constructor implements multivariate polynomials whose set of variables is `Symbol`. These variables do not commute. The only parameter of this construtor is the coefficient ring which may be non-commutative. However, coefficients and variables commute. The representation of the polynomials is recursive. The abbreviation for `XPolynomial` is `XPOLY`.

Other constructors like `XPolynomialRing`, `XRecursivePolynomial` as well as `XDistributedPolynomial`, `LiePolynomial` and `XPBWPolynomial` implement multivariate polynomials in non-commutative variables.

We illustrate now some of the facilities of the `XPOLY` domain constructor.

Define a polynomial ring over the integers.

```
poly := XPolynomial(Integer)
```

XPolynomial Integer

Type: Domain

Define a first polynomial,

```
pr:  poly := 2*x + 3*y-5
```

$$-5 + x\ 2 + y\ 3$$

Type: XPolynomial Integer

and a second one.

```
pr2:  poly := pr*pr
```

$$25 + x\ (-20 + x\ 4 + y\ 6) + y\ (-30 + x\ 6 + y\ 9)$$

Rewrite **pr** in a distributive way,

```
pd := expand pr
```

$$-5 + 2\ x + 3\ y$$

Type: XDistributedPolynomial(Symbol,Integer)

compute its square,

```
pd2 := pd*pd
```

$$25 - 20\ x - 30\ y + 4\ x^2 + 6\ x\ y + 6\ y\ x + 9\ y^2$$

Type: XDistributedPolynomial(Symbol,Integer)

and checks that:

```
expand(pr2) - pd2
```

$$0$$

Type: XDistributedPolynomial(Symbol,Integer)

We define:

```
qr := pr**3
```

$$-125 + x\ (150 + x\ (-60 + x\ 8 + y\ 12) + y\ (-90 + x\ 12 + y\ 18)) +$$

$$y\ (225 + x\ (-90 + x\ 12 + y\ 18) + y\ (-135 + x\ 18 + y\ 27))$$

Type: XPolynomial Integer

and:

```
qd := pd**3
```

$$-125 + 150\ x + 225\ y - 60\ x^2 - 90\ x\ y - 90\ y\ x - 135\ y^2 + 8\ x^3 + 12\ x^2\ y +$$

$$12\ x\ y\ x + 18\ x\ y^2 + 12\ y\ x^2 + 18\ y\ x\ y + 18\ y^2\ x + 27\ y^3$$

<div align="center">Type: XDistributedPolynomial(Symbol,Integer)</div>

We truncate **qd** at degree **3**:

```
trunc(qd,2)
```

$$-125 + 150\ x + 225\ y - 60\ x^2 - 90\ x\ y - 90\ y\ x - 135\ y^2$$

<div align="center">Type: XDistributedPolynomial(Symbol,Integer)</div>

The same for **qr**:

```
trunc(qr,2)
```

$$-125 + x\ (150 + x\ (-60) + y\ (-90)) + y\ (225 + x\ (-90) + y\ (-135))$$

<div align="right">Type: XPolynomial Integer</div>

We define:

```
Word := OrderedFreeMonoid Symbol
```

<div align="center">OrderedFreeMonoid Symbol</div>

<div align="right">Type: Domain</div>

and:

```
w:  Word := x*y**2
```

$$x\ y^2$$

<div align="right">Type: OrderedFreeMonoid Symbol</div>

The we can compute the right-quotient of **qr** by **r**:

```
rquo(qr,w)
```

<div align="center">18</div>

<div align="right">Type: XPolynomial Integer</div>

and the shuffle-product of **pr** by **r**:

```
sh(pr,w::poly)
```

$$x\ (x\ y\ y\ 4 + y\ (x\ y\ 2 + y\ (-5 + x\ 2 + y\ 9))) + y\ x\ y\ y\ 3$$

<div align="right">Type: XPolynomial Integer</div>

9.90 XPolynomialRing

The `XPolynomialRing` domain constructor implements generalized polynomials with coefficients from an arbitrary `Ring` (not necessarily commutative) and whose exponents are words from an arbitrary `OrderedMonoid` (not necessarily commutative too). Thus these polynomials are (finite) linear combinations of words.

This constructor takes two arguments. The first one is a `Ring` and the second is an `OrderedMonoid`. The abbreviation for `XPolynomialRing` is `XPR`.

Other constructors like `XPolynomial`, `XRecursivePolynomial` `XDistributedPolynomial`, `LiePolynomial` and `XPBWPolynomial` implement multivariate polynomials in non-commutative variables.

We illustrate now some of the facilities of the `XPR` domain constructor.

Define the free ordered monoid generated by the symbols.

```
Word := OrderedFreeMonoid(Symbol)
```

OrderedFreeMonoid Symbol

                                        Type: Domain

Define the linear combinations of these words with integer coefficients.

```
poly:= XPR(Integer,Word)
```

XPolynomialRing(Integer,OrderedFreeMonoid Symbol)

                                        Type: Domain

Then we define a first element from **poly**.

```
p:poly := 2 * x - 3 * y + 1
```

$$1 + 2\ x - 3\ y$$

        Type: XPolynomialRing(Integer,OrderedFreeMonoid Symbol)

And a second one.

```
q:poly := 2 * x + 1
```

$$1 + 2\ x$$

                    Type: XPolynomialRing(Integer,OrderedFreeMonoid Symbol)

We compute their sum,

```
p + q
```

$$2 + 4\ x - 3\ y$$

                    Type: XPolynomialRing(Integer,OrderedFreeMonoid Symbol)

their product,

```
p * q
```

$$1 + 4\ x - 3\ y + 4\ x^2 - 6\ y\ x$$

                    Type: XPolynomialRing(Integer,OrderedFreeMonoid Symbol)

and see that variables do not commute.

```
(p+q)**2-p**2-q**2-2*p*q
```

$$-6\ x\ y + 6\ y\ x$$

                    Type: XPolynomialRing(Integer,OrderedFreeMonoid Symbol)

Now we define a ring of square matrices,

```
M := SquareMatrix(2,Fraction Integer)
```

                    SquareMatrix(2,Fraction Integer)

                                                        Type: Domain

and the linear combinations of words with these matrices as coefficients.

```
poly1:= XPR(M,Word)
```

            XPolynomialRing(SquareMatrix(2, FractionInteger),
               OrderedFreeMonoidSymbol)

                                                        Type: Domain

Define a first matrix,

```
m1:M := matrix [ [i*j**2 for i in 1..2] for j in 1..2]
```

$$\begin{bmatrix} 1 & 2 \\ 4 & 8 \end{bmatrix}$$

Type: SquareMatrix(2,Fraction Integer)

a second one,

```
m2:M := m1 - 5/4
```

$$\begin{bmatrix} -\frac{1}{4} & 2 \\ 4 & \frac{27}{4} \end{bmatrix}$$

Type: SquareMatrix(2,Fraction Integer)

and a third one.

```
m3:  M := m2**2
```

$$\begin{bmatrix} \frac{129}{16} & 13 \\ 26 & \frac{857}{16} \end{bmatrix}$$

Type: SquareMatrix(2,Fraction Integer)

Define a polynomial,

```
pm:poly1 := m1*x + m2*y + m3*z - 2/3
```

$$\begin{bmatrix} -\frac{2}{3} & 0 \\ 0 & -\frac{2}{3} \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 4 & 8 \end{bmatrix} x + \begin{bmatrix} -\frac{1}{4} & 2 \\ 4 & \frac{27}{4} \end{bmatrix} y + \begin{bmatrix} \frac{129}{16} & 13 \\ 26 & \frac{857}{16} \end{bmatrix} z$$

Type: XPolynomialRing(SquareMatrix(2,Fraction Integer),
OrderedFreeMonoid Symbol)

a second one,

```
qm:poly1 := pm - m1*x
```

$$\begin{bmatrix} -\frac{2}{3} & 0 \\ 0 & -\frac{2}{3} \end{bmatrix} + \begin{bmatrix} -\frac{1}{4} & 2 \\ 4 & \frac{27}{4} \end{bmatrix} y + \begin{bmatrix} \frac{129}{16} & 13 \\ 26 & \frac{857}{16} \end{bmatrix} z$$

<div align="right">

Type: XPolynomialRing(SquareMatrix(2,Fraction Integer),
OrderedFreeMonoid Symbol)

</div>

and the following power.

`qm**3`

$$
\begin{bmatrix} -\frac{8}{27} & 0 \\ 0 & -\frac{8}{27} \end{bmatrix} + \begin{bmatrix} -\frac{1}{3} & \frac{8}{3} \\ \frac{16}{3} & 9 \end{bmatrix} y + \begin{bmatrix} \frac{43}{4} & \frac{52}{3} \\ \frac{104}{3} & \frac{857}{12} \end{bmatrix} z +
$$

$$
\begin{bmatrix} -\frac{129}{8} & -26 \\ -52 & -\frac{857}{8} \end{bmatrix} y^2 + \begin{bmatrix} -\frac{3199}{32} & -\frac{831}{4} \\ -\frac{831}{2} & -\frac{26467}{32} \end{bmatrix} y\,z + \begin{bmatrix} -\frac{3199}{32} & -\frac{831}{4} \\ -\frac{831}{2} & -\frac{26467}{32} \end{bmatrix} z\,y +
$$

$$
\begin{bmatrix} -\frac{103169}{128} & -\frac{6409}{4} \\ -\frac{6409}{2} & -\frac{820977}{128} \end{bmatrix} z^2 + \begin{bmatrix} \frac{3199}{64} & \frac{831}{8} \\ \frac{831}{4} & \frac{26467}{64} \end{bmatrix} y^3 +
$$

$$
\begin{bmatrix} \frac{103169}{256} & \frac{6409}{8} \\ \frac{6409}{4} & \frac{820977}{256} \end{bmatrix} y^2\,z + \begin{bmatrix} \frac{103169}{256} & \frac{6409}{8} \\ \frac{6409}{4} & \frac{820977}{256} \end{bmatrix} y\,z\,y +
$$

$$
\begin{bmatrix} \frac{3178239}{1024} & \frac{795341}{128} \\ \frac{795341}{64} & \frac{25447787}{1024} \end{bmatrix} y\,z^2 + \begin{bmatrix} \frac{103169}{256} & \frac{6409}{8} \\ \frac{6409}{4} & \frac{820977}{256} \end{bmatrix} z\,y^2 +
$$

$$
\begin{bmatrix} \frac{3178239}{1024} & \frac{795341}{128} \\ \frac{795341}{64} & \frac{25447787}{1024} \end{bmatrix} z\,y\,z + \begin{bmatrix} \frac{3178239}{1024} & \frac{795341}{128} \\ \frac{795341}{64} & \frac{25447787}{1024} \end{bmatrix} z^2\,y +
$$

$$
\begin{bmatrix} \frac{98625409}{4096} & \frac{12326223}{256} \\ \frac{12326223}{128} & \frac{788893897}{4096} \end{bmatrix} z^3
$$

<div align="right">

Type: XPolynomialRing(SquareMatrix(2,Fraction
Integer),OrderedFreeMonoid Symbol)

</div>

9.91   ZeroDimensionalSolvePackage

The `ZeroDimensionalSolvePackage` package constructor provides operations for computing symbolically the complex or real roots of zero-dimensional algebraic systems.

The package provides **no** multiplicity information (i.e. some returned roots may be double or higher) but only distinct roots are returned.

Complex roots are given by means of univariate representations of irreducible regular chains. These representations are computed by the **univariateSolve** operation (by calling the `InternalRationalUnivariateRepresentationPackage` package constructor which does the job).

Real roots are given by means of tuples of coordinates lying in the `RealClosure` of the coefficient ring. They are computed by the **realSolve** and **positiveSolve** operations. The former computes all the solutions of the input system with real coordinates whereas the later concentrate on the solutions with (strictly) positive coordinates. In both cases, the computations are performed by the `RealClosure` constructor.

Both computations of complex roots and real roots rely on triangular decompositions. These decompositions can be computed in two different ways. First, by a applying the **zeroSetSplit** operation from the `REGSET` domain constructor. In that case, no Groebner bases are computed. This strategy is used by default. Secondly, by applying the **zeroSetSplit** from `LEXTRIPK`. To use this later strategy with the operations **univariateSolve**, **realSolve** and **positiveSolve** one just needs to use an extra boolean argument.

Note that the way of understanding triangular decompositions is detailed in the example of the `RegularTriangularSet` constructor.

The `ZeroDimensionalSolvePackage` constructor takes three arguments. The first one **R** is the coefficient ring; it must belong to the categories `OrderedRing`, `EuclideanDomain`, `CharacteristicZero` and `RealConstant`. This means essentially that **R** is `Integer` or `Fraction(Integer)`. The second argument **ls** is the list of variables involved in the systems to solve. The third one MUST BE **concat(ls,s)** where **s** is an additional symbol used for the univariate representations. The abbreviation for `ZeroDimensionalSolvePackage` is `ZDSOLVE`.

We illustrate now how to use the constructor `ZDSOLVE` by two examples: the *Arnborg and Lazard* system and the *L-3* system (Aubry and Moreno Maza). Note that the use of this package is also demonstrated in the example of the `LexTriangularPackage` constructor.

Define the coefficient ring.

```
R := Integer
```

$$Integer$$

<div align="right">Type: Domain</div>

Define the lists of variables:

```
ls :  List Symbol := [x,y,z,t]
```

$$[x, y, z, t]$$

<div align="right">Type: List Symbol</div>

and:

```
ls2 :   List Symbol := [x,y,z,t,new()$Symbol]
```

$$[x, y, z, t, \%A]$$

<div align="right">Type: List Symbol</div>

Call the package:

```
pack := ZDSOLVE(R,ls,ls2)
```

$$ZeroDimensionalSolvePackage(Integer, [x, y, z, t], [x, y, z, t,$$

<div align="right">Type: Domain</div>

Define a polynomial system (Arnborg-Lazard)

```
p1 := x**2*y*z + x*y**2*z + x*y*z**2 + x*y*z + x*y + x*z + y*z
```

$$x \ y \ z^2 + \left(x \ y^2 + \left(x^2 + x + 1 \right) \ y + x \right) \ z + x \ y$$

<div align="right">Type: Polynomial Integer</div>

```
p2 := x**2*y**2*z + x*y**2*z**2 + x**2*y*z + x*y*z + y*z + x + z
```

$$x \ y^2 \ z^2 + \left(x^2 \ y^2 + \left(x^2 + x + 1 \right) \ y + 1 \right) \ z + x$$

<div align="right">Type: Polynomial Integer</div>

```
p3 := x**2*y**2*z**2 + x**2*y**2*z + x*y**2*z + x*y*z + x*z + z +
1
```

$$x^2 \ y^2 \ z^2 + \left(\left(x^2 + x \right) \ y^2 + x \ y + x + 1 \right) \ z + 1$$

<div align="right">Type: Polynomial Integer</div>

```
lp := [p1, p2, p3]
```

$$\left[x \ y \ z^2 + \left(x \ y^2 + \left(x^2 + x + 1 \right) \ y + x \right) \ z + x \ y, \right.$$
$$x \ y^2 \ z^2 + \left(x^2 \ y^2 + \left(x^2 + x + 1 \right) \ y + 1 \right) \ z + x,$$
$$\left. x^2 \ y^2 \ z^2 + \left(\left(x^2 + x \right) \ y^2 + x \ y + x + 1 \right) \ z + 1 \right]$$

```
                                    Type:  List Polynomial Integer
```

Note that these polynomials do not involve the variable **t**; we will use it in the second example.

First compute a decomposition into regular chains (i.e. regular triangular sets).

```
triangSolve(lp)$pack
```

$$\Big[\Big\{z^{20} - 6\ z^{19} - 41\ z^{18} + 71\ z^{17} + 106\ z^{16} + 92\ z^{15} + 197\ z^{14}+$$

$$145\ z^{13} + 257\ z^{12} + 278\ z^{11} + 201\ z^{10} + 278\ z^{9} + 257\ z^{8} + 145\ z^{7}+$$

$$197\ z^{6} + 92\ z^{5} + 106\ z^{4} + 71\ z^{3} - 41\ z^{2} - 6\ z + 1,$$

$$\big(14745844\ z^{19} + 50357474\ z^{18} - 130948857\ z^{17} - 185261586\ z^{16}-$$

$$180077775\ z^{15} - 338007307\ z^{14} - 275379623\ z^{13} - 453190404\ z^{12}-$$

$$474597456\ z^{11} - 366147695\ z^{10} - 481433567\ z^{9} - 430613166\ z^{8}-$$

$$261878358\ z^{7} - 326073537\ z^{6} - 163008796\ z^{5} - 177213227\ z^{4}-$$

$$104356755\ z^{3} + 65241699\ z^{2} + 9237732\ z - 1567348\big)\ y+$$

$$1917314\ z^{19} + 6508991\ z^{18} - 16973165\ z^{17} - 24000259\ z^{16}-$$

$$23349192\ z^{15} - 43786426\ z^{14} - 35696474\ z^{13} - 58724172\ z^{12}-$$

$$61480792\ z^{11} - 47452440\ z^{10} - 62378085\ z^{9} - 55776527\ z^{8}-$$

$$33940618\ z^{7} - 42233406\ z^{6} - 21122875\ z^{5} - 22958177\ z^{4}-$$

$$13504569\ z^{3} + 8448317\ z^{2} + 1195888\ z - 202934,$$

$$\big(\big(z^{3} - 2\ z\big)\ y^{2} + \big(-z^{3} - z^{2} - 2\ z - 1\big)\ y - z^{2} - z + 1\big)\ x + z^{2} - 1\Big\}\Big]$$

```
                       Type:  List RegularChain(Integer,[x,y,z,t])
```

We can see easily from this decomposition (consisting of a single regular chain) that the input system has 20 complex roots.

Then we compute a univariate representation of this regular chain.

```
univariateSolve(lp)$pack
```

$\left[\left[complexRoots = ?^{12} - 12\ ?^{11} + 24\ ?^{10} + 4\ ?^9 - 9\ ?^8 + 27\ ?^7 -\right.\right.$

$21\ ?^6 + 27\ ?^5 - 9\ ?^4 + 4\ ?^3 + 24\ ?^2 - 12\ ? + 1,$

$coordinates =$
$\left[63\ x + 62\ \%A^{11} - 721\ \%A^{10} + 1220\ \%A^9 + 705\ \%A^8 - 285\ \%A^7 +\right.$

$1512\ \%A^6 - 735\ \%A^5 + 1401\ \%A^4 - 21\ \%A^3 + 215\ \%A^2 + 1577\ \%A - 142,$

$63\ y - 75\ \%A^{11} + 890\ \%A^{10} - 1682\ \%A^9 - 516\ \%A^8 + 588\ \%A^7 - 1953\ \%A^6 +$

$1323\ \%A^5 - 1815\ \%A^4 + 426\ \%A^3 - 243\ \%A^2 - 1801\ \%A + 679,$

$\left.z - \%A]\right],$

$\left[complexRoots = ?^6 + ?^5 + ?^4 + ?^3 + ?^2 + ? + 1,\right.$
$coordinates = \left[x - \%A^5, y - \%A^3, z - \%A\right]\right],$

$\left[complexRoots = ?^2 + 5\ ? + 1, coordinates = [x - 1, y - 1, z - \%A]\right]\right]$

```
       Type:  List Record( complexRoots:  SparseUnivariatePolynomial
                   Integer, coordinates:  List Polynomial Integer)
```

We see that the zeros of our regular chain are split into three components. This is due to the use of univariate polynomial factorization.

Each of these components consist of two parts. The first one is an irreducible univariate polynomial **p(?)** which defines a simple algebraic extension of the field of fractions of **R**. The second one consists of multivariate polynomials **pol1(x,%A)**, **pol2(y,%A)** and **pol3(z,%A)**. Each of these polynomials involve two variables: one is an indeterminate **x**, **y** or **z** of the input system **lp** and the other is **%A** which represents any root of **p(?)**. Recall that this **%A** is the last element of the third parameter of ZDSOLVE. Thus any complex root **?** of **p(?)** leads to a solution of the input system **lp** by replacing **%A** by this **?** in **pol1(x,%A)**, **pol2(y,%A)** and **pol3(z,%A)**. Note that the polynomials **pol1(x,%A)**, **pol2(y,%A)** and **pol3(z,%A)** have degree one w.r.t. **x**, **y** or **z** respectively. This is always the case for all univariate representations. Hence the operation **univariateSolve** replaces a system of multivariate polynomials by a list of univariate polynomials, what justifies its name. Another example of univariate representations illustrates the `LexTriangularPackage` package constructor.

We now compute the solutions with real coordinates:

```
lr := realSolve(lp)$pack
```

$[[\%B1,$

$\frac{1184459}{1645371}\,\%B1^{19} - \frac{2335702}{548457}\,\%B1^{18} - \frac{5460230}{182819}\,\%B1^{17} + \frac{79900378}{1645371}\,\%B1^{16} + \frac{43953929}{548457}\,\%B1^{15} + \frac{13420192}{182819}\,\%B1^{14} + \frac{553986}{3731}\,\%B1^{13} + \frac{193381378}{1645371}\,\%B1^{12} + \frac{35978916}{182819}\,\%B1^{11} + \frac{358660781}{1645371}\,\%B1^{10} + \frac{271667666}{1645371}\,\%B1^{9} + \frac{118784873}{548457}\,\%B1^{8} + \frac{337505020}{1645371}\,\%B1^{7} + \frac{1389370}{11193}\,\%B1^{6} + \frac{688291}{4459}\,\%B1^{5} + \frac{3378002}{42189}\,\%B1^{4} + \frac{140671876}{1645371}\,\%B1^{3} + \frac{32325724}{548457}\,\%B1^{2} - \frac{8270}{343}\,\%B1 - \frac{9741532}{1645371},$

$-\frac{91729}{705159}\,\%B1^{19} + \frac{487915}{705159}\,\%B1^{18} + \frac{4114333}{705159}\,\%B1^{17} - \frac{1276987}{235053}\,\%B1^{16} - \frac{13243117}{705159}\,\%B1^{15} - \frac{16292173}{705159}\,\%B1^{14} - \frac{26536060}{705159}\,\%B1^{13} - \frac{722714}{18081}\,\%B1^{12} - \frac{5382578}{100737}\,\%B1^{11} - \frac{15449995}{235053}\,\%B1^{10} - \frac{14279770}{235053}\,\%B1^{9} - \frac{6603890}{100737}\,\%B1^{8} - \frac{409930}{6027}\,\%B1^{7} - \frac{37340389}{705159}\,\%B1^{6} - \frac{34893715}{705159}\,\%B1^{5} - \frac{26686318}{705159}\,\%B1^{4} - \frac{801511}{26117}\,\%B1^{3} - \frac{17206178}{705159}\,\%B1^{2} - \frac{4406102}{705159}\,\%B1 + \frac{377534}{705159}\Big],$

$[\%B2,$

$\frac{1184459}{1645371}\,\%B2^{19} - \frac{2335702}{548457}\,\%B2^{18} - \frac{5460230}{182819}\,\%B2^{17} + \frac{79900378}{1645371}\,\%B2^{16} + \frac{43953929}{548457}\,\%B2^{15} + \frac{13420192}{182819}\,\%B2^{14} + \frac{553986}{3731}\,\%B2^{13} + \frac{193381378}{1645371}\,\%B2^{12} + \frac{35978916}{182819}\,\%B2^{11} + \frac{358660781}{1645371}\,\%B2^{10} + \frac{271667666}{1645371}\,\%B2^{9} + \frac{118784873}{548457}\,\%B2^{8} + \frac{337505020}{1645371}\,\%B2^{7} + \frac{1389370}{11193}\,\%B2^{6} + \frac{688291}{4459}\,\%B2^{5} + \frac{3378002}{42189}\,\%B2^{4} + \frac{140671876}{1645371}\,\%B2^{3} + \frac{32325724}{548457}\,\%B2^{2} - \frac{8270}{343}\,\%B2 - \frac{9741532}{1645371},$

$-\frac{91729}{705159}\,\%B2^{19} + \frac{487915}{705159}\,\%B2^{18} + \frac{4114333}{705159}\,\%B2^{17} - \frac{1276987}{235053}\,\%B2^{16} - \frac{13243117}{705159}\,\%B2^{15} - \frac{16292173}{705159}\,\%B2^{14} - \frac{26536060}{705159}\,\%B2^{13} - \frac{722714}{18081}\,\%B2^{12} - \frac{5382578}{100737}\,\%B2^{11} - \frac{15449995}{235053}\,\%B2^{10} - \frac{14279770}{235053}\,\%B2^{9} - \frac{6603890}{100737}\,\%B2^{8} - \frac{409930}{6027}\,\%B2^{7} - \frac{37340389}{705159}\,\%B2^{6} - \frac{34893715}{705159}\,\%B2^{5} - \frac{26686318}{705159}\,\%B2^{4} - \frac{801511}{26117}\,\%B2^{3} - \frac{17206178}{705159}\,\%B2^{2} - \frac{4406102}{705159}\,\%B2 + \frac{377534}{705159}\Big],$

$[\%B3,$

$$\frac{1184459}{1645371}\,\%B3^{19} - \frac{2335702}{548457}\,\%B3^{18} - \frac{5460230}{182819}\,\%B3^{17} + \frac{79900378}{1645371}\,\%B3^{16} + \frac{43953929}{548457}\,\%B3^{15} + \frac{13420192}{182819}\,\%B3^{14} + \frac{553986}{3731}\,\%B3^{13} + \frac{193381378}{1645371}\,\%B3^{12} + \frac{35978916}{182819}\,\%B3^{11} + \frac{358660781}{1645371}\,\%B3^{10} + \frac{271667666}{1645371}\,\%B3^{9} + \frac{118784873}{548457}\,\%B3^{8} + \frac{337505020}{1645371}\,\%B3^{7} + \frac{1389370}{11193}\,\%B3^{6} + \frac{688291}{4459}\,\%B3^{5} + \frac{3378002}{42189}\,\%B3^{4} + \frac{140671876}{1645371}\,\%B3^{3} + \frac{32325724}{548457}\,\%B3^{2} - \frac{8270}{343}\,\%B3 - \frac{9741532}{1645371},$$

$$-\frac{91729}{705159}\,\%B3^{19} + \frac{487915}{705159}\,\%B3^{18} + \frac{4114333}{705159}\,\%B3^{17} - \frac{1276987}{235053}\,\%B3^{16} - \frac{13243117}{705159}\,\%B3^{15} - \frac{16292173}{705159}\,\%B3^{14} - \frac{26536060}{705159}\,\%B3^{13} - \frac{722714}{18081}\,\%B3^{12} - \frac{5382578}{100737}\,\%B3^{11} - \frac{15449995}{235053}\,\%B3^{10} - \frac{14279770}{235053}\,\%B3^{9} - \frac{6603890}{100737}\,\%B3^{8} - \frac{409930}{6027}\,\%B3^{7} - \frac{37340389}{705159}\,\%B3^{6} - \frac{34893715}{705159}\,\%B3^{5} - \frac{26686318}{705159}\,\%B3^{4} - \frac{801511}{26117}\,\%B3^{3} - \frac{17206178}{705159}\,\%B3^{2} - \frac{4406102}{705159}\,\%B3 + \frac{377534}{705159}\Big],$$

$[\%B4,$

$$\frac{1184459}{1645371}\,\%B4^{19} - \frac{2335702}{548457}\,\%B4^{18} - \frac{5460230}{182819}\,\%B4^{17} + \frac{79900378}{1645371}\,\%B4^{16} + \frac{43953929}{548457}\,\%B4^{15} + \frac{13420192}{182819}\,\%B4^{14} + \frac{553986}{3731}\,\%B4^{13} + \frac{193381378}{1645371}\,\%B4^{12} + \frac{35978916}{182819}\,\%B4^{11} + \frac{358660781}{1645371}\,\%B4^{10} + \frac{271667666}{1645371}\,\%B4^{9} + \frac{118784873}{548457}\,\%B4^{8} + \frac{337505020}{1645371}\,\%B4^{7} + \frac{1389370}{11193}\,\%B4^{6} + \frac{688291}{4459}\,\%B4^{5} + \frac{3378002}{42189}\,\%B4^{4} + \frac{140671876}{1645371}\,\%B4^{3} + \frac{32325724}{548457}\,\%B4^{2} - \frac{8270}{343}\,\%B4 - \frac{9741532}{1645371},$$

$$-\frac{91729}{705159}\,\%B4^{19} + \frac{487915}{705159}\,\%B4^{18} + \frac{4114333}{705159}\,\%B4^{17} - \frac{1276987}{235053}\,\%B4^{16} - \frac{13243117}{705159}\,\%B4^{15} - \frac{16292173}{705159}\,\%B4^{14} - \frac{26536060}{705159}\,\%B4^{13} - \frac{722714}{18081}\,\%B4^{12} - \frac{5382578}{100737}\,\%B4^{11} - \frac{15449995}{235053}\,\%B4^{10} - \frac{14279770}{235053}\,\%B4^{9} - \frac{6603890}{100737}\,\%B4^{8} - \frac{409930}{6027}\,\%B4^{7} - \frac{37340389}{705159}\,\%B4^{6} - \frac{34893715}{705159}\,\%B4^{5} - \frac{26686318}{705159}\,\%B4^{4} - \frac{801511}{26117}\,\%B4^{3} - \frac{17206178}{705159}\,\%B4^{2} - \frac{4406102}{705159}\,\%B4 + \frac{377534}{705159}\Big],$$

$[\%B5,$

$$\frac{1184459}{1645371}\,\%B5^{19} - \frac{2335702}{548457}\,\%B5^{18} - \frac{5460230}{182819}\,\%B5^{17} + \frac{79900378}{1645371}\,\%B5^{16} + \frac{43953929}{548457}\,\%B5^{15} + \frac{13420192}{182819}\,\%B5^{14} + \frac{553986}{3731}\,\%B5^{13} + \frac{193381378}{1645371}\,\%B5^{12} + \frac{35978916}{182819}\,\%B5^{11} + \frac{358660781}{1645371}\,\%B5^{10} + \frac{271667666}{1645371}\,\%B5^{9} + \frac{118784873}{548457}\,\%B5^{8} + \frac{337505020}{1645371}\,\%B5^{7} + \frac{1389370}{11193}\,\%B5^{6} + \frac{688291}{4459}\,\%B5^{5} + \frac{3378002}{42189}\,\%B5^{4} + \frac{140671876}{1645371}\,\%B5^{3} + \frac{32325724}{548457}\,\%B5^{2} - \frac{8270}{343}\,\%B5 - \frac{9741532}{1645371},$$

$$-\frac{91729}{705159}\,\%B5^{19} + \frac{487915}{705159}\,\%B5^{18} + \frac{4114333}{705159}\,\%B5^{17} - \frac{1276987}{235053}\,\%B5^{16} - \frac{13243117}{705159}\,\%B5^{15} - \frac{16292173}{705159}\,\%B5^{14} - \frac{26536060}{705159}\,\%B5^{13} - \frac{722714}{18081}\,\%B5^{12} - \frac{5382578}{100737}\,\%B5^{11} - \frac{15449995}{235053}\,\%B5^{10} - \frac{14279770}{235053}\,\%B5^{9} - \frac{6603890}{100737}\,\%B5^{8} - \frac{409930}{6027}\,\%B5^{7} - \frac{37340389}{705159}\,\%B5^{6} - \frac{34893715}{705159}\,\%B5^{5} - \frac{26686318}{705159}\,\%B5^{4} - \frac{801511}{26117}\,\%B5^{3} - \frac{17206178}{705159}\,\%B5^{2} - \frac{4406102}{705159}\,\%B5 + \frac{377534}{705159}\Bigg],$$

$[\%B6,$

$$\frac{1184459}{1645371}\,\%B6^{19} - \frac{2335702}{548457}\,\%B6^{18} - \frac{5460230}{182819}\,\%B6^{17} + \frac{79900378}{1645371}\,\%B6^{16} + \frac{43953929}{548457}\,\%B6^{15} + \frac{13420192}{182819}\,\%B6^{14} + \frac{553986}{3731}\,\%B6^{13} + \frac{193381378}{1645371}\,\%B6^{12} + \frac{35978916}{182819}\,\%B6^{11} + \frac{358660781}{1645371}\,\%B6^{10} + \frac{271667666}{1645371}\,\%B6^{9} + \frac{118784873}{548457}\,\%B6^{8} + \frac{337505020}{1645371}\,\%B6^{7} + \frac{1389370}{11193}\,\%B6^{6} + \frac{688291}{4459}\,\%B6^{5} + \frac{3378002}{42189}\,\%B6^{4} + \frac{140671876}{1645371}\,\%B6^{3} + \frac{32325724}{548457}\,\%B6^{2} - \frac{8270}{343}\,\%B6 - \frac{9741532}{1645371},$$

$$-\frac{91729}{705159}\,\%B6^{19} + \frac{487915}{705159}\,\%B6^{18} + \frac{4114333}{705159}\,\%B6^{17} - \frac{1276987}{235053}\,\%B6^{16} - \frac{13243117}{705159}\,\%B6^{15} - \frac{16292173}{705159}\,\%B6^{14} - \frac{26536060}{705159}\,\%B6^{13} - \frac{722714}{18081}\,\%B6^{12} - \frac{5382578}{100737}\,\%B6^{11} - \frac{15449995}{235053}\,\%B6^{10} - \frac{14279770}{235053}\,\%B6^{9} - \frac{6603890}{100737}\,\%B6^{8} - \frac{409930}{6027}\,\%B6^{7} - \frac{37340389}{705159}\,\%B6^{6} - \frac{34893715}{705159}\,\%B6^{5} - \frac{26686318}{705159}\,\%B6^{4} - \frac{801511}{26117}\,\%B6^{3} - \frac{17206178}{705159}\,\%B6^{2} - \frac{4406102}{705159}\,\%B6 + \frac{377534}{705159}\Bigg],$$

$[\%B7,$

$$\frac{1184459}{1645371}\%B7^{19} - \frac{2335702}{548457}\%B7^{18} - \frac{5460230}{182819}\%B7^{17} + \frac{79900378}{1645371}\%B7^{16} + \frac{43953929}{548457}\%B7^{15} + \frac{13420192}{182819}\%B7^{14} + \frac{553986}{3731}\%B7^{13} + \frac{193381378}{1645371}\%B7^{12} + \frac{35978916}{182819}\%B7^{11} + \frac{358660781}{1645371}\%B7^{10} + \frac{271667666}{1645371}\%B7^9 + \frac{118784873}{548457}\%B7^8 + \frac{337505020}{1645371}\%B7^7 + \frac{1389370}{11193}\%B7^6 + \frac{688291}{4459}\%B7^5 + \frac{3378002}{42189}\%B7^4 + \frac{140671876}{1645371}\%B7^3 + \frac{32325724}{548457}\%B7^2 - \frac{8270}{343}\%B7 - \frac{9741532}{1645371},$$

$$-\frac{91729}{705159}\%B7^{19} + \frac{487915}{705159}\%B7^{18} + \frac{4114333}{705159}\%B7^{17} - \frac{1276987}{235053}\%B7^{16} - \frac{13243117}{705159}\%B7^{15} - \frac{16292173}{705159}\%B7^{14} - \frac{26536060}{705159}\%B7^{13} - \frac{722714}{18081}\%B7^{12} - \frac{5382578}{100737}\%B7^{11} - \frac{15449995}{235053}\%B7^{10} - \frac{14279770}{235053}\%B7^9 - \frac{6603890}{100737}\%B7^8 - \frac{409930}{6027}\%B7^7 - \frac{37340389}{705159}\%B7^6 - \frac{34893715}{705159}\%B7^5 - \frac{26686318}{705159}\%B7^4 - \frac{801511}{26117}\%B7^3 - \frac{17206178}{705159}\%B7^2 - \frac{4406102}{705159}\%B7 + \frac{377534}{705159}\Big],$$

$[\%B8,$

$$\frac{1184459}{1645371}\%B8^{19} - \frac{2335702}{548457}\%B8^{18} - \frac{5460230}{182819}\%B8^{17} + \frac{79900378}{1645371}\%B8^{16} + \frac{43953929}{548457}\%B8^{15} + \frac{13420192}{182819}\%B8^{14} + \frac{553986}{3731}\%B8^{13} + \frac{193381378}{1645371}\%B8^{12} + \frac{35978916}{182819}\%B8^{11} + \frac{358660781}{1645371}\%B8^{10} + \frac{271667666}{1645371}\%B8^9 + \frac{118784873}{548457}\%B8^8 + \frac{337505020}{1645371}\%B8^7 + \frac{1389370}{11193}\%B8^6 + \frac{688291}{4459}\%B8^5 + \frac{3378002}{42189}\%B8^4 + \frac{140671876}{1645371}\%B8^3 + \frac{32325724}{548457}\%B8^2 - \frac{8270}{343}\%B8 - \frac{9741532}{1645371},$$

$$-\frac{91729}{705159}\%B8^{19} + \frac{487915}{705159}\%B8^{18} + \frac{4114333}{705159}\%B8^{17} - \frac{1276987}{235053}\%B8^{16} - \frac{13243117}{705159}\%B8^{15} - \frac{16292173}{705159}\%B8^{14} - \frac{26536060}{705159}\%B8^{13} - \frac{722714}{18081}\%B8^{12} - \frac{5382578}{100737}\%B8^{11} - \frac{15449995}{235053}\%B8^{10} - \frac{14279770}{235053}\%B8^9 - \frac{6603890}{100737}\%B8^8 - \frac{409930}{6027}\%B8^7 - \frac{37340389}{705159}\%B8^6 - \frac{34893715}{705159}\%B8^5 - \frac{26686318}{705159}\%B8^4 - \frac{801511}{26117}\%B8^3 - \frac{17206178}{705159}\%B8^2 - \frac{4406102}{705159}\%B8 + \frac{377534}{705159}\Big]\Big]\Big]$$

```
                        Type:  List List RealClosure Fraction Integer
```

The number of real solutions for the input system is:

```
# lr
```

8

<div align="right">

```
Type:  PositiveInteger
```

</div>

Each of these real solutions is given by a list of elements in `RealClosure(R)`. In these 8 lists, the first element is a value of **z**, the second of **y** and the last of **x**. This is logical since by setting the list of variables of the package to [**x,y,z,t**] we mean that the elimination ordering on the variables is **t ¡ z ¡ y ¡ x** . Note that each system treated by the `ZDSOLVE` package constructor needs only to be zero-dimensional w.r.t. the variables involved in the system it-self and not necessarily w.r.t. all the variables used to define the package.

We can approximate these real numbers as follows. This computation takes between 30 sec. and 5 min, depending on your machine.

```
[ [approximate(r,1/1000000) for r in point] for point in lr]
```

$\left[\left[-\frac{10048059}{2097152},\right.\right.$

$$\left(\frac{\begin{matrix}4503057316985387943524397913838966414596731976211768219335881208388\\5516314058924567176091423629695777403099833360761048898228916578137\\0943098385973311372025848469391323761570195067603576011659174549868\\1538209878909485152342039281129312614132985654697714546466149548782\\5919941188447041722440491921567263542158028061437758844364634410045\\25302478656192316328821417 5\end{matrix}}{\begin{matrix}4503057283025245488516511806985826635083100693757320465280554706865\\6449495775099168672018894380904083548179317185938627976245515189835\\7079304877442429148870882984032418920030143612331486020082144373379\\0755311243632918648954217042289495712900161194988079570236638654430\\69392027148979688266712323356043491523434068924275280417338574817381\\189277066143312396681216,\end{matrix}}\right.$$

$$\left.\frac{\begin{matrix}2106260768823475073894798680486016596249607148690685538763683715020\\6396808586496507900558895056468933094470970999378021873290953258987\\8524724902071750498366048207515661873872451468533306001120296463516\\6381351543255982200250305283981086837110614842307026091211297929876\\89628568183047905476005638076266490561846205530604781619178201158870\\37891389881895\end{matrix}}{\begin{matrix}2106260609498464192472113804816474175341962953296434102413903142368\\7579676852738885855909759652117788621898728819539436402462973570619\\5981232610365979902512686325867656720234210687703171018424748418142\\3288921837681237062708470295706218485928867400771937828499200923760\\593314168901000666373896347598118228556731037072026474496776228383762\\9939232800768\end{matrix}}\right)\right],$$

$\left[-\frac{2563013}{2097152},\right.$

$$\left(\frac{\begin{array}{l}-2611346176791927789698617693237757719238259963063541781922752330\\4401898996680729283384907686235932074421259259867338159322435048092\\94838752303023733723680666816744617300172727135331157124289 7\end{array}}{\begin{array}{l}1165225400505222530583981916004589143757226610276858990008790134819\\914940922413753983971394019523433320408139928153188829495755455163\\963417619308395977544797140231469234269034921938055593984},\end{array}}\right.$$

$$\left.\frac{\begin{array}{l}357259455027591722109658872961578827299851705467560323957819814100\\603409173528282659062190230446696394197103892330452627332931637375\\7450061978989228611097699708725046623537 3\end{array}}{\begin{array}{l}1039548269345598936877071244834026055800814551120170592200522366591\\7594096594864423391410294529502651799899601048118758225302053465051\\315812439017247289173865014702966308864}\end{array}}\right],$$

$\left[-\frac{1715967}{2097152},\right.$

$$\left(\frac{\begin{array}{l}-4213093533784303521084839517977082390377261503969586224828998436\\6060306560763593745648137734983766031212678225658014362069395199514\\65182225805246972874100225439524 91\end{array}}{\begin{array}{l}9441814144185374458649692034349224052436597470966253663930641960795\\8058825854931998401916999176594432648246411351873835838881478673401\\9307857605820364195856822304768},\end{array}}\right.$$

$$\left.\frac{\begin{array}{l}7635833347112644222515625424410831225347475669008589338834162172501\\9049994376346730876809042845208919919925302105720971453918982731389\\0725914035\end{array}}{\begin{array}{l}2624188764086097199784297610478066633934230467895851602278580978503\\7845492057884990196406022669660268915801035435676250390186298871412\\8491675648}\end{array}}\right],$$

$\left[-\frac{437701}{2097152},\right.$

$$\left(\frac{\begin{array}{l}1683106908638349588322172332654225913562986313181951031452750161441\\4974734553281507213648683555796467816035077771990750778352133664845\\33654913836237413047 59\end{array}}{\begin{array}{l}1683106868095213389001709982705913638963077668731226111167785188004\\9074252262986803258878109626141402985973669842648879989083770687999\\9845423381649008099328},\end{array}}\right.$$

$$\left.\frac{\begin{array}{l}4961550109835010186422681013422108735958714801003760639707968096646\\9128267084728344431172391721910424921345096631241113 3\end{array}}{\begin{array}{l}4961549872757738315509192078210209029852897118611097126236384040829\\37659261914313170254867464792718363492160482442215424}\end{array}}\right],$$

$$\left[\frac{222801}{2097152},\right.$$

$$\left(\frac{\begin{array}{l}-899488488040242826510759512197069142713604569254197827557300186\\5213759921588137716696126349101655220195142994932299137183241705 86\\7672383477\end{array}}{\begin{array}{l}11678899986650263721777651006918885827089699602299347696908357524\\5707777941643520947376786650776940588894276458771854243425562 59924\\56372224\end{array}},\right.$$

$$\left.\frac{\begin{array}{l}-23897048881331568783208015443738083956127715092084910198474529\\91885509546519525467839016613593999693886640036283570552321155037 87\\1291458703265\end{array}}{\begin{array}{l}53554872736450963260904032866899319059882254446854114332215938336\\8119295756283367146865429034074699365628592559911760212044618344 31\\45479421952\end{array}}\right],$$

$$\left[\frac{765693}{2097152},\right.$$

$$\left(\frac{\begin{array}{l}855896921981671626787324476117819808872469895861667014021376575 4\\3220023032516857861186783308402033288376543395234187049177495183 40\\77251289900039100963037 3148561\end{array}}{\begin{array}{l}29414424455330107909764284113763934998155802159458569179064525354\\95723013856818941702330228779890141296236721138154231997238917322 1\\5671196524446393317194601 59488\end{array}},\right.$$

$$\left.\frac{\begin{array}{l}-20576182305825721012476503248602425611113025815435888088439236 6\\2767549382241659362712290777612800192921420574408948085193743688 58\\27622246433251878894899015\end{array}}{\begin{array}{l}26715982033257355380979523535014502205763137598908350970917225206\\4271019877190266718394890628986371475967836029248394920461647153 77\\7777532418066109536665 6\end{array}}\right],$$

$$\left[\frac{5743879}{2097152},\right.$$

$$\left(\frac{\begin{array}{l}10762888169689068479555463947735702081714567249426186140236631 23\\57476896085043426397139807254659277266215883344979769861745539788 7\\56290007298476800060834355318980169340872720504761255988923275756 3\\83052868895353542180948277105891754260289006094194962087408300785 8\\36666945350176624841488732463225\end{array}}{\begin{array}{l}31317689570803179466484619400235520441903766134585849862285496319\\16196601616219781765615532532294746529648276430583810894079374566 4\\60757823146888581195556029208515218838883200318658407469399426063 2\\60589828612309231596669129707986481319851571942927230340622934023 9\\2348670304206815304408450990 08\end{array}},\right.$$

$$\left[\begin{pmatrix} -2113286699185750918364120475565458437870172489865485994389828135335264444665284557526492734931691731407872701432935503473348172076098720545849008780077564160534317894688366119529739980502944162668550098127961950496210221942878089359674925850594427768502251789758706752831632503615 \\ \hline 162761558493798758024290662434710458088914446616845971804315383940837252553330980807036369958550221601121108710326360955102602776941408739114812622116813978168258743807532259146613193997545720052234983856896428563444801856203827237878735446010610614151801093561720517063962536181766 \end{pmatrix} \right],$$

$$\left[\frac{19739877}{2097152}, \right.$$

$$\begin{pmatrix} -299724993683270330379901580486152094921504038750070717770128576672019253057942247895356602435986014310154780163808277161116037221287484777803580987284314922548423836585801362934170532170258233350918009601789937023985935304900460493389738370308534103470899088808148539811320184645824588006153947707416994872958759602107502158919488144768548710315309312954673321901337026710982009022823005107518607185928457030277807397796525813862762239286996106809728023675 \\ \hline 230843327485227859072891008119181102390650414132143264612393679487393331927060896070213819341764789836062022951917663293763178685145501476602720625902225250555174182368889688380663660257443176047224029209319672947516024726883412114189331884872866184443492728728511289708076755286489505658586403317856591038706500611280151640352274103736099055605447694952705922707080959304949125751955470887925959552929920110858560812556635485429471554031675979542656381353984, \end{pmatrix}$$

$$\left[\begin{pmatrix} -512818926354822848909627639786894008060093841066308045940796633584500926410949052045982531625008472301004703502449743652303892581895928931293158470135392762143543439867426304729390912285013385199069649023156609437199433379507078262401172758774998929661127731837229462420711653791043655457414608288470130554391262041935488541073594015777589660282236457586461183151294397397471516692046506185060376287516256195847052412587282839139194642913955 \\ \hline 2288281939778439330531208793181290471183631092455368990386390824243509463644236249773080647438987739144921607794682653851741189091711741868145114978337284191822497675868358729486644730856622552687209203724411800481405702837198310642291275676195774614443815996713502629391749783590041470860127752372996488627742672487622480063268808889324891850842494934347337603075939980268208482904859678177751444657499798278726169630532176732017172372520966 \end{pmatrix} \right] \right]$$

Type: List List Fraction Integer

We can also concentrate on the solutions with real (strictly) positive coordinates:

```
lpr := positiveSolve(lp)$pack
```

$$[\,]$$

                    Type: List List RealClosure Fraction Integer

Thus we have checked that the input system has no solution with strictly positive coordinates.

Let us define another polynomial system (*L-3*).

```
f0 := x**3 + y + z + t- 1
```

$$z + y + x^3 + t - 1$$

                              Type: Polynomial Integer

```
f1 := x + y**3 + z + t -1
```

$$z + y^3 + x + t - 1$$

                              Type: Polynomial Integer

```
f2 := x + y + z**3 + t-1
```

$$z^3 + y + x + t - 1$$

                              Type: Polynomial Integer

```
f3 := x + y + z + t**3 -1
```

$$z + y + x + t^3 - 1$$

                              Type: Polynomial Integer

```
lf := [f0, f1, f2, f3]
```

$$\left[z + y + x^3 + t - 1, z + y^3 + x + t - 1, \right.$$
$$\left. z^3 + y + x + t - 1, z + y + x + t^3 - 1 \right]$$

First compute a decomposition into regular chains (i.e. regular triangular sets).

```
lts := triangSolve(lf)$pack
```

$\Big[\big\{t^2 + t + 1, z^3 - z - t^3 + t, \left(3\ z + 3\ t^3 - 3\right)\ y^2 + \left(3\ z^2 + \left(6\ t^3 - 6\right)\ z + 3\ t^6 -\right.$
$6\ t^3 + 3\big)\ y + \left(3\ t^3 - 3\right)\ z^2 + \left(3\ t^6 - 6\ t^3 + 3\right)\ z + t^9 - 3\ t^6 + 5\ t^3 - 3\ t,$
$x + y + z\big\}, \big\{t^{16} - 6\ t^{13} + 9\ t^{10} + 4\ t^7 + 15\ t^4 - 54\ t^2 + 27,$

$\left(4907232\ t^{15} + 40893984\ t^{14} - 115013088\ t^{13} + 22805712\ t^{12} + 36330336\ t^{11} +\right.$
$162959040\ t^{10} - 159859440\ t^9 - 156802608\ t^8 + 117168768\ t^7 +$
$126282384\ t^6 - 129351600\ t^5 + 306646992\ t^4 + 475302816\ t^3 -$
$1006837776\ t^2 - 237269088\ t + 480716208\big)\ z +$

$48\ t^{54} - 912\ t^{51} + 8232\ t^{48} - 72\ t^{46} - 46848\ t^{45} + 1152\ t^{43} + 186324\ t^{42} -$
$3780\ t^{40} - 543144\ t^{39} - 3168\ t^{38} - 21384\ t^{37} + 1175251\ t^{36} + 41184\ t^{35} +$
$278003\ t^{34} - 1843242\ t^{33} - 301815\ t^{32} - 1440726\ t^{31} + 1912012\ t^{30} +$
$1442826\ t^{29} + 4696262\ t^{28} - 922481\ t^{27} - 4816188\ t^{26} - 10583524\ t^{25} -$
$208751\ t^{24} + 11472138\ t^{23} + 16762859\ t^{22} - 857663\ t^{21} - 19328175\ t^{20} -$
$18270421\ t^{19} + 4914903\ t^{18} + 22483044\ t^{17} + 12926517\ t^{16} - 8605511\ t^{15} -$
$17455518\ t^{14} - 5014597\ t^{13} + 8108814\ t^{12} + 8465535\ t^{11} + 190542\ t^{10} -$
$4305624\ t^9 - 2226123\ t^8 + 661905\ t^7 + 1169775\ t^6 + 226260\ t^5 -$
$209952\ t^4 - 141183\ t^3 + 27216\ t,$

$\left(3\ z + 3\ t^3 - 3\right)\ y^2 + \left(3\ z^2 + \left(6\ t^3 - 6\right)\ z + 3\ t^6 - 6\ t^3 + 3\right)\ y + \left(3\ t^3 - 3\right)\ z^2 +$
$\left(3\ t^6 - 6\ t^3 + 3\right)\ z + t^9 - 3\ t^6 + 5\ t^3 - 3\ t, x + y + z + t^3 - 1\big\},$
$\big\{t, z - 1, y^2 - 1, x + y\big\}, \big\{t - 1, z, y^2 - 1, x + y\big\}, \big\{t - 1, z^2 - 1, z\ y + 1, x\big\},$
$\big\{t^{16} - 6\ t^{13} + 9\ t^{10} + 4\ t^7 + 15\ t^4 - 54\ t^2 + 27,$

$\left(4907232\ t^{29} + 40893984\ t^{28} - 115013088\ t^{27} - 1730448\ t^{26} - 168139584\ t^{25} +\right.$
$738024480\ t^{24} - 195372288\ t^{23} + 315849456\ t^{22} - 2567279232\ t^{21} +$
$937147968\ t^{20} + 1026357696\ t^{19} + 4780488240\ t^{18} - 2893767696\ t^{17} -$
$5617160352\ t^{16} - 3427651728\ t^{15} + 5001100848\ t^{14} + 8720098416\ t^{13} +$
$2331732960\ t^{12} - 499046544\ t^{11} - 16243306272\ t^{10} - 9748123200\ t^9 +$
$3927244320\ t^8 + 25257280896\ t^7 + 10348032096\ t^6 - 17128672128\ t^5 -$
$14755488768\ t^4 + 544086720\ t^3 + 10848188736\ t^2 + 1423614528\ t -$
$2884297248\big)\ z -$

$48\ t^{68} + 1152\ t^{65} - 13560\ t^{62} + 360\ t^{60} + 103656\ t^{59} - 7560\ t^{57} - 572820\ t^{56} +$
$71316\ t^{54} + 2414556\ t^{53} + 2736\ t^{52} - 402876\ t^{51} - 7985131\ t^{50} - 49248\ t^{49} +$
$1431133\ t^{48} + 20977409\ t^{47} + 521487\ t^{46} - 2697635\ t^{45} - 43763654\ t^{44} -$
$3756573\ t^{43} - 2093410\ t^{42} + 71546495\ t^{41} + 19699032\ t^{40} + 35025028\ t^{39} -$
$89623786\ t^{38} - 77798760\ t^{37} - 138654191\ t^{36} + 87596128\ t^{35} + 235642497\ t^{34} +$
$349607642\ t^{33} - 93299834\ t^{32} - 551563167\ t^{31} - 630995176\ t^{30} +$
$186818962\ t^{29} + 995427468\ t^{28} + 828416204\ t^{27} - 393919231\ t^{26} -$
$1076617485\ t^{25} - 1609479791\ t^{24} + 595738126\ t^{23} + 1198787136\ t^{22} +$
$4342832069\ t^{21} - 2075938757\ t^{20} - 4390835799\ t^{19} - 4822843033\ t^{18} +$
$6932747678\ t^{17} + 6172196808\ t^{16} + 1141517740\ t^{15} - 4981677585\ t^{14} -$
$9819815280\ t^{13} - 7404299976\ t^{12} - 157295760\ t^{11} + 29124027630\ t^{10} +$
$14856038208\ t^{9} - 16184101410\ t^{8} - 26935440354\ t^{7} - 3574164258\ t^{6} +$
$10271338974\ t^{5} + 11191425264\ t^{4} + 6869861262\ t^{3} - 9780477840\ t^{2} -$
$3586674168\ t + 2884297248,$

$\left(3\ z^3 + \left(6\ t^3 - 6\right)\ z^2 + \left(6\ t^6 - 12\ t^3 + 3\right)\ z + 2\ t^9 - 6\ t^6 + t^3 + 3\ t\right)\ y +$

$\left(3\ t^3 - 3\right)\ z^3 + \left(6\ t^6 - 12\ t^3 + 6\right)\ z^2 + \left(4\ t^9 - 12\ t^6 + 11\ t^3 - 3\right)\ z +$

$t^{12} - 4\ t^9 + 5\ t^6 - 2\ t^3, x + y + z + t^3 - 1\big\},$

$\left\{t - 1, z^2 - 1, y, x + z\right\}, \left\{t^8 + t^7 + t^6 - 2\ t^5 - 2\ t^4 - 2\ t^3 + 19\ t^2 + 19\ t - 8,\right.$

$\left(2395770\ t^7 + 3934440\ t^6 - 3902067\ t^5 - 10084164\ t^4 - 1010448\ t^3 + 32386932\ t^2 +\right.$

$22413225\ t - 10432368\big)\ z - 463519\ t^7 + 3586833\ t^6 + 9494955\ t^5 - 8539305\ t^4 -$

$33283098\ t^3 + 35479377\ t^2 + 46263256\ t - 17419896,$

$\left(3\ z^4 + \left(9\ t^3 - 9\right)\ z^3 + \left(12\ t^6 - 24\ t^3 + 9\right)\ z^2 + \left(-152\ t^3 + 219\ t - 67\right)\ z -\right.$

$41\ t^6 + 57\ t^4 + 25\ t^3 - 57\ t + 16\big)\ y + \left(3\ t^3 - 3\right)\ z^4 + \left(9\ t^6 - 18\ t^3 + 9\right)\ z^3 +$

$\left(-181\ t^3 + 270\ t - 89\right)\ z^2 + \left(-92\ t^6 + 135\ t^4 + 49\ t^3 - 135\ t + 43\right)\ z +$

$27\ t^7 - 27\ t^6 - 54\ t^4 + 396\ t^3 - 486\ t + 144, x + y + z + t^3 - 1\big\},$

$\left\{t, z - t^3 + 1, y - 1, x - 1\right\}, \{t - 1, z, y, x\}, \{t, z - 1, y, x\}, \{t, z, y - 1, x\},$

$\{t, z, y, x - 1\}]$

<div align="center">Type: List RegularChain(Integer,[x,y,z,t])</div>

Then we compute a univariate representation.

```
univariateSolve(lf)$pack
```

$[[complexRoots =?, coordinates = [x-1, y-1, z+1, t-\%A]],$
$[complexRoots =?, coordinates = [x, y-1, z, t-\%A]],$
$[complexRoots = ?-1, coordinates = [x, y, z, t-\%A]],$
$[complexRoots =?, coordinates = [x-1, y, z, t-\%A]],$
$[complexRoots =?, coordinates = [x, y, z-1, t-\%A]],$
$[complexRoots = ?-2, coordinates = [x-1, y+1, z, t-1]],$
$[complexRoots =?, coordinates = [x+1, y-1, z, t-1]],$
$[complexRoots = ?-1, coordinates = [x-1, y+1, z-1, t]],$
$[complexRoots = ?+1, coordinates = [x+1, y-1, z-1, t]],$
$\big[complexroots = ?^6 - 2\ ?^3 + 3\ ?^2 - 3, coordinates = \big[2\ x + \%A^3 + \%A - 1,$
$2\ y + \%A^3 + \%A - 1, z - \%A, t - \%A\big]\big]\,,$
$\big[complexRoots = ?^5 + 3\ ?^3 - 2\ ?^2 + 3\ ? - 3, coordinates = [x - \%A,$
$y - \%A, z + \%A^3 + 2\ \%A - 1, t - \%A]\big]\,,$
$\big[complexRoots = ?^4 - ?^3 - 2\ ?^2 + 3, coordinates = \big[x + \%A^3 - \%A - 1,$
$y + \%A^3 - \%A - 1, z - \%A^3 + 2\ \%A + 1, t - \%A\big]\big]\,,$
$\big[complexRoots = ? + 1, coordinates = [x - 1, y - 1, z, t - \%A]\big]\,,$
$\big[complexRoots = ?^6 + 2\ ?^3 + 3\ ?^2 - 3, coordinates = \big[2\ x - \%A^3 - \%A - 1,$
$y + \%A, 2\ z - \%A^3 - \%A - 1, t + \%A\big]\big]\,,$
$\big[complexRoots = ?^6 + 12\ ?^4 + 20\ ?^3 - 45\ ?^2 - 42\ ? - 953, coordinates =$
$\big[12609\ x + 23\ \%A^5 + 49\ \%A^4 - 46\ \%A^3 + 362\ \%A^2 - 5015\ \%A - 8239,$
$25218\ y + 23\ \%A^5 + 49\ \%A^4 - 46\ \%A^3 + 362\ \%A^2 + 7594\ \%A - 8239,$
$25218\ z + 23\ \%A^5 + 49\ \%A^4 - 46\ \%A^3 + 362\ \%A^2 + 7594\ \%A - 8239,$
$12609\ t + 23\ \%A^5 + 49\ \%A^4 - 46\ \%A^3 + 362\ \%A^2 - 5015\ \%A - 8239\big]\big]\,,$
$\big[complexRoots = ?^5 + 12\ ?^3 - 16\ ?^2 + 48\ ? - 96, coordinates = \big[8\ x + \%A^3 +$
$8\ \%A - 8, 2\ y - \%A, 2\ z - \%A, 2\ t - \%A]\big]\,,$
$\big[complexRoots = ?^5 + ?^4 - 5\ ?^3 - 3\ ?^2 + 9\ ? + 3, coordinates = \big[2\ x - \%A^3 +$
$2\ \%A - 1, 2\ y + \%A^3 - 4\ \%A + 1, 2\ z - \%A^3 + 2\ \%A - 1, 2\ t - \%A^3 + 2\ \%A - 1]\big]\,,$
$\big[complexRoots = ?^4 - 3\ ?^3 + 4\ ?^2 - 6\ ? + 13, coordinates = \big[9\ x - 2\ \%A^3 +$
$4\ \%A^2 - \%A + 2, 9\ y + \%A^3 - 2\ \%A^2 + 5\ \%A - 1, 9\ z + \%A^3 - 2\ \%A^2 +$
$5\ \%A - 1, 9\ t + \%A^3 - 2\ \%A^2 - 4\ \%A - 1]\big]\,,$
$\big[complexRoots = ?^4 - 11\ ?^2 + 37, coordinates = \big[3\ x - \%A^2 + 7, 6\ y + \%A^2 +$
$3\ \%A - 7, 3\ z - \%A^2 + 7, 6\ t + \%A^2 - 3\ \%A - 7]\big]\,,$
$[complexRoots = ? + 1, coordinates = [x - 1, y, z - 1, t + 1]],$
$[complexRoots = ? + 2, coordinates = [x, y - 1, z - 1, t + 1]],$
$[complexRoots = ? - 2, coordinates = [x, y - 1, z + 1, t - 1]],$
$[complexRoots =?, coordinates = [x, y + 1, z - 1, t - 1]],$
$[complexRoots = ? - 2, coordinates = [x - 1, y, z + 1, t - 1]],$
$[complexRoots =?, coordinates = [x + 1, y, z - 1, t - 1]],$
$\big[complexRoots = ?^4 + 5\ ?^3 + 16\ ?^2 + 30\ ? + 57, coordinates = \big[151\ x + 15\ \%A^3 +$
$54\ \%A^2 + 104\ \%A + 93, 151\ y - 10\ \%A^3 - 36\ \%A^2 - 19\ \%A - 62,$
$151\ z - 5\ \%A^3 - 18\ \%A^2 - 85\ \%A - 31, 151\ t - 5\ \%A^3 - 18\ \%A^2 - 85\ \%A - 31]\big]\,,$
$\big[complexRoots = ?^4 - ?^3 - 2\ ?^2 + 3, coordinates = \big[x - \%A^3 + 2\ \%A + 1,$
$y + \%A^3 - \%A - 1, z - \%A, t + \%A^3 - \%A - 1]\big]\,,$
$\big[complexRoots = ?^4 + 2\ ?^3 - 8\ ?^2 + 48, coordinates = \big[8\ x - \%A^3 + 4\ \%A - 8,$
$2\ y + \%A, 8\ z + \%A^3 - 8\ \%A + 8, 8\ t - \%A^3 + 4\ \%A - 8]\big]\,,$
$\big[complexRoots = ?^5 + ?^4 - 2\ ?^3 - 4\ ?^2 + 5\ ? + 8,$
$coordinates = \big[3\ x + \%A^3 - 1, 3\ y + \%A^3 - 1, 3\ z + \%A^3 - 1, t - \%A]\big]\,,$
$\big[complexRoots = ?^3 + 3\ ? - 1, coordinates = [x - \%A, y - \%A, z - \%A, t - \%A]\big]\big]$

```
     Type:  List Record( complexRoots:  SparseUnivariatePolynomial
                 Integer, coordinates:  List Polynomial Integer)
```

Note that this computation is made from the input system **lf**.

However it is possible to reuse a pre-computed regular chain as follows:

```
ts := lts.1
```

$$\left\{ t^2 + t + 1, z^3 - z - t^3 + t, \right.$$

$$\left(3\ z + 3\ t^3 - 3 \right)\ y^2 + \left(3\ z^2 + \left(6\ t^3 - 6 \right)\ z + 3\ t^6 - 6\ t^3 + 3 \right)\ y+$$

$$\left(3\ t^3 - 3 \right)\ z^2 + \left(3\ t^6 - 6\ t^3 + 3 \right)\ z + t^9 - 3\ t^6 + 5\ t^3 - 3\ t, x + y + z \right\}$$

```
                      Type:  RegularChain(Integer,[x,y,z,t])
```

```
univariateSolve(ts)$pack
```

$$\left[\left[complexRoots = ?^4 + 5\ ?^3 + 16\ ?^2 + 30\ ? + 57, p \right.\right.$$
$$coordinates = \left[151\ x + 15\ \%A^3 + 54\ \%A^2 + 104\ \%A + 93, \right.$$
$$151\ y - 10\ \%A^3 - 36\ \%A^2 - 19\ \%A - 62,$$
$$151\ z - 5\ \%A^3 - 18\ \%A^2 - 85\ \%A - 31,$$
$$\left. \left. 151\ t - 5\ \%A^3 - 18\ \%A^2 - 85\ \%A - 31 \right] \right],$$

$$\left[complexRoots = ?^4 - ?^3 - 2\ ?^2 + 3, \right.$$
$$coordinates = \left[x - \%A^3 + 2\ \%A + 1, y + \%A^3 - \%A - 1, \right.$$
$$\left. \left. z - \%A, t + \%A^3 - \%A - 1 \right] \right],$$

$$\left[complexRoots = ?^4 + 2\ ?^3 - 8\ ?^2 + 48, \right.$$
$$coordinates = \left[8\ x - \%A^3 + 4\ \%A - 8, 2\ y + \%A, \right.$$
$$\left. \left. \left. 8\ z + \%A^3 - 8\ \%A + 8, 8\ t - \%A^3 + 4\ \%A - 8 \right] \right] \right]$$

```
     Type:  List Record( complexRoots:  SparseUnivariatePolynomial
                 Integer, coordinates:  List Polynomial Integer)
```

```
realSolve(ts)$pack
```

$$[]$$

```
                      Type:  List List RealClosure Fraction Integer
```

We compute now the full set of points with real coordinates:

```
lr2 := realSolve(lf)$pack
```

$[[0, -1, 1, 1], [0, 0, 1, 0], [1, 0, 0, 0], [0, 0, 0, 1], [0, 1, 0, 0],$

$[1, 0, \%B37, -\%B37], [1, 0, \%B38, -\%B38],$

$[0, 1, \%B35, -\%B35], [0, 1, \%B36, -\%B36], [-1, 0, 1, 1],$

$\Big[\%B32, \frac{1}{27}\ \%B32^{15} + \frac{2}{27}\ \%B32^{14} + \frac{1}{27}\ \%B32^{13} - \frac{4}{27}\ \%B32^{12} - \frac{11}{27}\ \%B32^{11} -$

$\frac{4}{27}\ \%B32^{10} + \frac{1}{27}\ \%B32^{9} + \frac{14}{27}\ \%B32^{8} + \frac{1}{27}\ \%B32^{7} + \frac{2}{9}\ \%B32^{6} +$

$\frac{1}{3}\ \%B32^{5} + \frac{2}{9}\ \%B32^{4} + \%B32^{3} + \frac{4}{3}\ \%B32^{2} - \%B32 - 2,$

$-\frac{1}{54}\ \%B32^{15} - \qquad \frac{1}{27}\ \%B32^{14} - \frac{1}{54}\ \%B32^{13} + \frac{2}{27}\ \%B32^{12} + \frac{11}{54}\ \%B32^{11} +$

$\frac{2}{27}\ \%B32^{10} - \frac{1}{54}\ \%B32^{9} - \frac{7}{27}\ \%B32^{8} - \frac{1}{54}\ \%B32^{7} - \frac{1}{9}\ \%B32^{6} -$

$\frac{1}{6}\ \%B32^{5} - \frac{1}{9}\ \%B32^{4} - \%B32^{3} - \frac{2}{3}\ \%B32^{2} + \frac{1}{2}\ \%B32 + \frac{3}{2},$

$-\frac{1}{54}\ \%B32^{15} - \qquad \frac{1}{27}\ \%B32^{14} - \frac{1}{54}\ \%B32^{13} + \frac{2}{27}\ \%B32^{12} + \frac{11}{54}\ \%B32^{11} +$

$\frac{2}{27}\ \%B32^{10} - \frac{1}{54}\ \%B32^{9} - \frac{7}{27}\ \%B32^{8} - \frac{1}{54}\ \%B32^{7} - \frac{1}{9}\ \%B32^{6} -$

$\frac{1}{6}\ \%B32^{5} - \frac{1}{9}\ \%B32^{4} - \%B32^{3} - \frac{2}{3}\ \%B32^{2} + \frac{1}{2}\ \%B32 + \frac{3}{2}\Big],$

$\Big[\%B33, \frac{1}{27}\ \%B33^{15} + \frac{2}{27}\ \%B33^{14} + \frac{1}{27}\ \%B33^{13} - \frac{4}{27}\ \%B33^{12} - \frac{11}{27}\ \%B33^{11} -$

$\frac{4}{27}\ \%B33^{10} + \frac{1}{27}\ \%B33^{9} + \frac{14}{27}\ \%B33^{8} + \frac{1}{27}\ \%B33^{7} + \frac{2}{9}\ \%B33^{6} +$

$\frac{1}{3}\ \%B33^{5} + \frac{2}{9}\ \%B33^{4} + \%B33^{3} + \frac{4}{3}\ \%B33^{2} - \%B33 - 2,$

$-\frac{1}{54}\ \%B33^{15} - \qquad \frac{1}{27}\ \%B33^{14} - \frac{1}{54}\ \%B33^{13} + \frac{2}{27}\ \%B33^{12} + \frac{11}{54}\ \%B33^{11} +$

$\frac{2}{27}\ \%B33^{10} - \frac{1}{54}\ \%B33^{9} - \frac{7}{27}\ \%B33^{8} - \frac{1}{54}\ \%B33^{7} - \frac{1}{9}\ \%B33^{6} -$

$\frac{1}{6}\ \%B33^{5} - \frac{1}{9}\ \%B33^{4} - \%B33^{3} - \frac{2}{3}\ \%B33^{2} + \frac{1}{2}\ \%B33 + \frac{3}{2},$

$$-\tfrac{1}{54}\ \%B33^{15}-\qquad \tfrac{1}{27}\ \%B33^{14}-\tfrac{1}{54}\ \%B33^{13}+\tfrac{2}{27}\ \%B33^{12}+\tfrac{11}{54}\ \%B33^{11}+$$

$$\tfrac{2}{27}\ \%B33^{10}-\tfrac{1}{54}\ \%B33^{9}-\tfrac{7}{27}\ \%B33^{8}-\tfrac{1}{54}\ \%B33^{7}-\tfrac{1}{9}\ \%B33^{6}-$$

$$\tfrac{1}{6}\ \%B33^{5}-\tfrac{1}{9}\ \%B33^{4}-\%B33^{3}-\tfrac{2}{3}\ \%B33^{2}+\tfrac{1}{2}\ \%B33+\tfrac{3}{2}\Big],$$

$$\Big[\%B34, \tfrac{1}{27}\ \%B34^{15}+\tfrac{2}{27}\ \%B34^{14}+\tfrac{1}{27}\ \%B34^{13}-\tfrac{4}{27}\ \%B34^{12}-\tfrac{11}{27}\ \%B34^{11}-$$

$$\tfrac{4}{27}\ \%B34^{10}+\tfrac{1}{27}\ \%B34^{9}+\tfrac{14}{27}\ \%B34^{8}+\tfrac{1}{27}\ \%B34^{7}+\tfrac{2}{9}\ \%B34^{6}+$$

$$\tfrac{1}{3}\ \%B34^{5}+\tfrac{2}{9}\ \%B34^{4}+\%B34^{3}+\tfrac{4}{3}\ \%B34^{2}-\%B34-2,$$

$$-\tfrac{1}{54}\ \%B34^{15}-\qquad \tfrac{1}{27}\ \%B34^{14}-\tfrac{1}{54}\ \%B34^{13}+\tfrac{2}{27}\ \%B34^{12}+\tfrac{11}{54}\ \%B34^{11}+$$

$$\tfrac{2}{27}\ \%B34^{10}-\tfrac{1}{54}\ \%B34^{9}-\tfrac{7}{27}\ \%B34^{8}-\tfrac{1}{54}\ \%B34^{7}-\tfrac{1}{9}\ \%B34^{6}-$$

$$\tfrac{1}{6}\ \%B34^{5}-\tfrac{1}{9}\ \%B34^{4}-\%B34^{3}-\tfrac{2}{3}\ \%B34^{2}+\tfrac{1}{2}\ \%B34+\tfrac{3}{2},$$

$$-\tfrac{1}{54}\ \%B34^{15}-\qquad \tfrac{1}{27}\ \%B34^{14}-\tfrac{1}{54}\ \%B34^{13}+\tfrac{2}{27}\ \%B34^{12}+\tfrac{11}{54}\ \%B34^{11}+$$

$$\tfrac{2}{27}\ \%B34^{10}-\tfrac{1}{54}\ \%B34^{9}-\tfrac{7}{27}\ \%B34^{8}-\tfrac{1}{54}\ \%B34^{7}-\tfrac{1}{9}\ \%B34^{6}-$$

$$\tfrac{1}{6}\ \%B34^{5}-\tfrac{1}{9}\ \%B34^{4}-\%B34^{3}-\tfrac{2}{3}\ \%B34^{2}+\tfrac{1}{2}\ \%B34+\tfrac{3}{2}\Big],$$

$$[-1,1,0,1],[-1,1,1,0],$$

$$\Big[\%B23, -\tfrac{1}{54}\ \%B23^{15}-\tfrac{1}{27}\ \%B23^{14}-\tfrac{1}{54}\ \%B23^{13}+\tfrac{2}{27}\ \%B23^{12}+\tfrac{11}{54}\ \%B23^{11}+$$

$$\tfrac{2}{27}\ \%B23^{10}-\tfrac{1}{54}\ \%B23^{9}-\tfrac{7}{27}\ \%B23^{8}-\tfrac{1}{54}\ \%B23^{7}-\tfrac{1}{9}\ \%B23^{6}-$$

$$\tfrac{1}{6}\ \%B23^{5}-\tfrac{1}{9}\ \%B23^{4}-\%B23^{3}-\tfrac{2}{3}\ \%B23^{2}+\tfrac{1}{2}\ \%B23+\tfrac{3}{2},$$

$$\%B30, -\%B30+\tfrac{1}{54}\ \%B23^{15}+\qquad \tfrac{1}{27}\ \%B23^{14}+\tfrac{1}{54}\ \%B23^{13}-\tfrac{2}{27}\ \%B23^{12}-\tfrac{11}{54}\ \%B23^{11}-$$

$$\tfrac{2}{27}\ \%B23^{10}+\tfrac{1}{54}\ \%B23^{9}+\tfrac{7}{27}\ \%B23^{8}+\tfrac{1}{54}\ \%B23^{7}+\tfrac{1}{9}\ \%B23^{6}+$$

$$\tfrac{1}{6}\ \%B23^{5}+\tfrac{1}{9}\ \%B23^{4}+\tfrac{2}{3}\ \%B23^{2}-\tfrac{1}{2}\ \%B23-\tfrac{1}{2}\Big],$$

$\Big[\%B23, -\frac{1}{54}\ \%B23^{15} - \frac{1}{27}\ \%B23^{14} - \frac{1}{54}\ \%B23^{13} + \frac{2}{27}\ \%B23^{12} + \frac{11}{54}\ \%B23^{11} +$

$\frac{2}{27}\ \%B23^{10} - \frac{1}{54}\ \%B23^{9} - \frac{7}{27}\ \%B23^{8} - \frac{1}{54}\ \%B23^{7} - \frac{1}{9}\ \%B23^{6} -$

$\frac{1}{6}\ \%B23^{5} - \frac{1}{9}\ \%B23^{4} - \%B23^{3} - \frac{2}{3}\ \%B23^{2} + \frac{1}{2}\ \%B23 + \frac{3}{2},$

$\%B31, -\%B31 + \frac{1}{54}\ \%B23^{15} + \frac{1}{27}\ \%B23^{14} + \frac{1}{54}\ \%B23^{13} - \frac{2}{27}\ \%B23^{12} -$

$\frac{11}{54}\ \%B23^{11} - \frac{2}{27}\ \%B23^{10} + \frac{1}{54}\ \%B23^{9} + \frac{7}{27}\ \%B23^{8} + \frac{1}{54}\ \%B23^{7} +$

$\frac{1}{9}\ \%B23^{6} + \frac{1}{6}\ \%B23^{5} + \frac{1}{9}\ \%B23^{4} + \frac{2}{3}\ \%B23^{2} - \frac{1}{2}\ \%B23 - \frac{1}{2}\Big],$

$\Big[\%B24, -\frac{1}{54}\ \%B24^{15} - \frac{1}{27}\ \%B24^{14} - \frac{1}{54}\ \%B24^{13} + \frac{2}{27}\ \%B24^{12} + \frac{11}{54}\ \%B24^{11} +$

$\frac{2}{27}\ \%B24^{10} - \frac{1}{54}\ \%B24^{9} - \frac{7}{27}\ \%B24^{8} - \frac{1}{54}\ \%B24^{7} - \frac{1}{9}\ \%B24^{6} -$

$\frac{1}{6}\ \%B24^{5} - \frac{1}{9}\ \%B24^{4} - \%B24^{3} - \frac{2}{3}\ \%B24^{2} + \frac{1}{2}\ \%B24 + \frac{3}{2},$

$\%B28, -\%B28 + \frac{1}{54}\ \%B24^{15} + \frac{1}{27}\ \%B24^{14} + \frac{1}{54}\ \%B24^{13} - \frac{2}{27}\ \%B24^{12} - \frac{11}{54}\ \%B24^{11} -$

$\frac{2}{27}\ \%B24^{10} + \frac{1}{54}\ \%B24^{9} + \frac{7}{27}\ \%B24^{8} + \frac{1}{54}\ \%B24^{7} + \frac{1}{9}\ \%B24^{6} +$

$\frac{1}{6}\ \%B24^{5} + \frac{1}{9}\ \%B24^{4} + \frac{2}{3}\ \%B24^{2} - \frac{1}{2}\ \%B24 - \frac{1}{2}\Big],$

$\Big[\%B24, -\frac{1}{54}\ \%B24^{15} - \frac{1}{27}\ \%B24^{14} - \frac{1}{54}\ \%B24^{13} + \frac{2}{27}\ \%B24^{12} + \frac{11}{54}\ \%B24^{11} +$

$\frac{2}{27}\ \%B24^{10} - \frac{1}{54}\ \%B24^{9} - \frac{7}{27}\ \%B24^{8} - \frac{1}{54}\ \%B24^{7} - \frac{1}{9}\ \%B24^{6} -$

$\frac{1}{6}\ \%B24^{5} - \frac{1}{9}\ \%B24^{4} - \%B24^{3} - \frac{2}{3}\ \%B24^{2} + \frac{1}{2}\ \%B24 + \frac{3}{2},$

$\%B29, -\%B29 + \frac{1}{54}\ \%B24^{15} + \frac{1}{27}\ \%B24^{14} + \frac{1}{54}\ \%B24^{13} - \frac{2}{27}\ \%B24^{12} - \frac{11}{54}\ \%B24^{11} -$

$\frac{2}{27}\ \%B24^{10} + \frac{1}{54}\ \%B24^{9} + \frac{7}{27}\ \%B24^{8} + \frac{1}{54}\ \%B24^{7} + \frac{1}{9}\ \%B24^{6} +$

$\frac{1}{6}\ \%B24^{5} + \frac{1}{9}\ \%B24^{4} + \frac{2}{3}\ \%B24^{2} - \frac{1}{2}\ \%B24 - \frac{1}{2}\Big],$

$$\Big[\%B25, -\tfrac{1}{54}\,\%B25^{15} - \tfrac{1}{27}\,\%B25^{14} - \tfrac{1}{54}\,\%B25^{13} + \tfrac{2}{27}\,\%B25^{12} + \tfrac{11}{54}\,\%B25^{11} +$$

$$\tfrac{2}{27}\,\%B25^{10} - \tfrac{1}{54}\,\%B25^{9} - \tfrac{7}{27}\,\%B25^{8} - \tfrac{1}{54}\,\%B25^{7} - \tfrac{1}{9}\,\%B25^{6} -$$

$$\tfrac{1}{6}\,\%B25^{5} - \tfrac{1}{9}\,\%B25^{4} - \%B25^{3} - \tfrac{2}{3}\,\%B25^{2} + \tfrac{1}{2}\,\%B25 + \tfrac{3}{2},$$

$$\%B26, -\%B26 + \tfrac{1}{54}\,\%B25^{15} + \tfrac{1}{27}\,\%B25^{14} + \tfrac{1}{54}\,\%B25^{13} - \tfrac{2}{27}\,\%B25^{12} - \tfrac{11}{54}\,\%B25^{11} -$$

$$\tfrac{2}{27}\,\%B25^{10} + \tfrac{1}{54}\,\%B25^{9} + \tfrac{7}{27}\,\%B25^{8} + \tfrac{1}{54}\,\%B25^{7} + \tfrac{1}{9}\,\%B25^{6} +$$

$$\tfrac{1}{6}\,\%B25^{5} + \tfrac{1}{9}\,\%B25^{4} + \tfrac{2}{3}\,\%B25^{2} - \tfrac{1}{2}\,\%B25 - \tfrac{1}{2}\Big],$$

$$\Big[\%B25, -\tfrac{1}{54}\,\%B25^{15} - \tfrac{1}{27}\,\%B25^{14} - \tfrac{1}{54}\,\%B25^{13} + \tfrac{2}{27}\,\%B25^{12} + \tfrac{11}{54}\,\%B25^{11} +$$

$$\tfrac{2}{27}\,\%B25^{10} - \tfrac{1}{54}\,\%B25^{9} - \tfrac{7}{27}\,\%B25^{8} - \tfrac{1}{54}\,\%B25^{7} - \tfrac{1}{9}\,\%B25^{6} -$$

$$\tfrac{1}{6}\,\%B25^{5} - \tfrac{1}{9}\,\%B25^{4} - \%B25^{3} - \tfrac{2}{3}\,\%B25^{2} + \tfrac{1}{2}\,\%B25 + \tfrac{3}{2},$$

$$\%B27, -\%B27 + \tfrac{1}{54}\,\%B25^{15} + \tfrac{1}{27}\,\%B25^{14} + \tfrac{1}{54}\,\%B25^{13} - \tfrac{2}{27}\,\%B25^{12} - \tfrac{11}{54}\,\%B25^{11} -$$

$$\tfrac{2}{27}\,\%B25^{10} + \tfrac{1}{54}\,\%B25^{9} + \tfrac{7}{27}\,\%B25^{8} + \tfrac{1}{54}\,\%B25^{7} + \tfrac{1}{9}\,\%B25^{6} +$$

$$\tfrac{1}{6}\,\%B25^{5} + \tfrac{1}{9}\,\%B25^{4} + \tfrac{2}{3}\,\%B25^{2} - \tfrac{1}{2}\,\%B25 - \tfrac{1}{2}\Big],$$

$$[1, \%B21, -\%B21, 0], [1, \%B22, -\%B22, 0], [1, \%B19, 0, -\%B19], [1, \%B20, 0, -\%B20],$$

$$\Big[\%B17, -\tfrac{1}{3}\,\%B17^{3} + \tfrac{1}{3}, -\tfrac{1}{3}\,\%B17^{3} + \tfrac{1}{3}, -\tfrac{1}{3}\,\%B17^{3} + \tfrac{1}{3}\Big],$$

$$\Big[\%B18, -\tfrac{1}{3}\,\%B18^{3} + \tfrac{1}{3}, -\tfrac{1}{3}\,\%B18^{3} + \tfrac{1}{3}, -\tfrac{1}{3}\,\%B18^{3} + \tfrac{1}{3}\Big]\Big]$$

                    Type: List List RealClosure Fraction Integer

The number of real solutions for the input system is:

```
#lr2
```

27

                                        Type: PositiveInteger

Another example of computation of real solutions illustrates the
`LexTriangularPackage` package constructor.

We concentrate now on the solutions with real (strictly) positive coordinates:

`lpr2 := positiveSolve(lf)$pack`

$$\left[\left[\%B40, -\frac{1}{3}\ \%B40^3 + \frac{1}{3}, -\frac{1}{3}\ \%B40^3 + \frac{1}{3}, -\frac{1}{3}\ \%B40^3 + \frac{1}{3}\right]\right]$$

              Type: List List RealClosure Fraction Integer

Finally, we approximate the coordinates of this point with 20 exact digits:

`[approximate(r,1/10**21)::Float for r in lpr2.1]`

$$[0.3221853546\ 2608559291, 0.3221853546\ 2608559291,$$
$$0.3221853546\ 2608559291, 0.32218535462608559291]$$

                                        Type: List Float

Chapter 10

Interactive Programming

Programming in the interpreter is easy. So is the use of Axiom's graphics facility. Both are rather flexible and allow you to use them for many interesting applications. However, both require learning some basic ideas and skills.

All graphics examples in the gallery section are either produced directly by interactive commands or by interpreter programs. Four of these programs are introduced here. By the end of this chapter you will know enough about graphics and programming in the interpreter to not only understand all these examples, but to tackle interesting and difficult problems on your own. The appendix on graphics lists all the remaining commands and programs used to create these images.

10.1   Drawing Ribbons Interactively

We begin our discussion of interactive graphics with the creation of a useful facility: plotting ribbons of two-graphs in three-space. Suppose you want to draw the two-dimensional graphs of n functions $f_i(x), 1 \leq i \leq n$, all over some fixed range of x. One approach is to create a two-dimensional graph for each one, then superpose one on top of the other. What you will more than likely get is a jumbled mess. Even if you make each function a different color, the result is likely to be confusing.

A better approach is to display each of the $f_i(x)$ in three dimensions as a "ribbon" of some appropriate width along the y-direction, laying down each ribbon next to the previous one. A ribbon is simply a function of x and y depending only on x.

We illustrate this for $f_i(x)$ defined as simple powers of x for x ranging between -1 and 1.

Draw the ribbon for $z = x^2$.

```
draw(x**2,x=-1..1,y=0..1)
```

Now that was easy! What you get is a "wire-mesh" rendition of the ribbon. That's fine for now. Notice that the mesh-size is small in both the x and the y directions. Axiom normally computes points in both these directions. This is unnecessary. One step is all we need in the y-direction. To have Axiom economize on y-points, we re-draw the ribbon with option $var2Steps == 1$.

Re-draw the ribbon, but with option $var2Steps == 1$ so that only 1 step is computed in the y direction.

```
vp := draw(x**2,x=-1..1,y=0..1,var2Steps==1)
```

The operation has created a viewport, that is, a graphics window on your screen. We assigned the viewport to vp and now we manipulate its contents.

Graphs are objects, like numbers and algebraic expressions. You may want to do some experimenting with graphs. For example, say

```
showRegion(vp, "on")
```

to put a bounding box around the ribbon. Try it! Issue $rotate(vp, -45, 90)$ to rotate the figure -45 longitudinal degrees and 90 latitudinal degrees.

Here is a different rotation. This turns the graph so you can view it along the y-axis.

```
rotate(vp, 0, -90)
```

There are many other things you can do. In fact, most everything you can do interactively using the three-dimensional control panel (such as translating, zooming, resizing, coloring, perspective and lighting selections) can also be done directly by operations (see Chapter 7 on page 297 for more details).

When you are done experimenting, say $reset(vp)$ to restore the picture to its original position and settings.

Let's add another ribbon to our picture—one for x^3. Since y ranges from 0 to 1 for the first ribbon, now let y range from 1 to 2. This puts the second ribbon next to the first one.

How do you add a second ribbon to the viewport? One method is to extract the "space" component from the viewport using the operation **subspace**. You can think of the space component as the object inside the window (here, the ribbon). Let's call it sp. To add the second ribbon, you draw the second ribbon using the option $space == sp$.

Extract the space component of vp.

```
sp := subspace(vp)
```

Add the ribbon for x^3 alongside that for x^2.

```
vp := draw(x**3,x=-1..1,y=1..2,var2Steps==1, space==sp)
```

Unless you moved the original viewport, the new viewport covers the old one. You might want to check that the old object is still there by moving the top window.

Let's show quadrilateral polygon outlines on the ribbons and then enclose the ribbons in a box.

Show quadrilateral polygon outlines.

```
drawStyle(vp,"shade");outlineRender(vp,"on")
```

Enclose the ribbons in a box.

```
rotate(vp,20,-60); showRegion(vp,"on")
```

This process has become tedious! If we had to add two or three more ribbons, we would have to repeat the above steps several more times. It is time to write an interpreter program to help us take care of the details.

10.2   A Ribbon Program

The above approach creates a new viewport for each additional ribbon. A better approach is to build one object composed of all ribbons before creating a viewport. To do this, use **makeObject** rather than **draw**. The operations have similar formats, but **draw** returns a viewport and **makeObject** returns a space object.

We now create a function **drawRibbons** of two arguments: *flist*, a list of formulas for the ribbons you want to draw, and *xrange*, the range over which you want them drawn. Using this function, you can just say

```
drawRibbons([x**2, x**3], x=-1..1)
```

to do all of the work required in the last section. Here is the **drawRibbons** program. Invoke your favorite editor and create a file called **ribbon.input** containing the following program.

Here are some remarks on the syntax used in the **drawRibbons** function (consult Chapter 6 on page 231 for more details). Unlike most other programming languages which use semicolons, parentheses, or *begin–end* brackets to delineate the structure of programs, the structure of an Axiom program is determined by

```
drawRibbons(flist, xrange) ==}{}
  sp := createThreeSpace()                     Create empty space $sp$.
  y0 := 0                                      The initial ribbon position.
  for f in flist repeat                        For each function $f$,
    makeObject(f, xrange, y=y0..y0+1,          create and add a ribbon
      space==sp, var2Steps == 1)               for $f$ to the space $sp$.
    y0 := y0 + 1                               The next ribbon position.
  vp := makeViewport3D(sp, "Ribbons")          Create viewport.
  drawStyle(vp, "shade")                       Select shading style.
  outlineRender(vp, "on")                      Show polygon outlines.
  showRegion(vp,"on")                          Enclose in a box.
  n := \# flist                                The number of ribbons
  zoom(vp,n,1,n)                               Zoom in x- and z-directions.
  rotate(vp,0,75)                              Change the angle of view.
  vp                                           Return the viewport.
```

Figure 10.1: The first **drawRibbons** function.

indentation. The first line of the function definition always begins in column 1. All other lines of the function are indented with respect to the first line and form a *pile* (see 5.2 on page 199).

The definition of **drawRibbons** consists of a pile of expressions to be executed one after another. Each expression of the pile is indented at the same level. Lines 4-7 designate one single expression: since lines 5-7 are indented with respect to the others, these lines are treated as a continuation of line 4. Also since lines 5 and 7 have the same indentation level, these lines designate a pile within the outer pile.

The last line of a pile usually gives the value returned by the pile. Here it is also the value returned by the function. Axiom knows this is the last line of the function because it is the last line of the file. In other cases, a new expression beginning in column one signals the end of a function.

The line **drawStyle(vp,"shade")** is given after the viewport has been created to select the draw style. We have also used the **zoom** option. Without the zoom, the viewport region would be scaled equally in all three coordinate directions.

Let's try the function **drawRibbons**. First you must read the file to give Axiom the function definition.

Read the input file.

```
)read ribbon
```

Draw ribbons for $x, x^2, \ldots, x^5$ for $-1 \le x \le 1$

```
drawRibbons([x**i for i in 1..5],x=-1..1)
```

```
drawRibbons(flist, xrange, yrange) ==}{}
  sp := createThreeSpace()                    Create empty space $sp$.
  num := \# flist                             The number of ribbons.
  yVar := variable yrange                      The ribbon variable.
  y0:Float    := lo segment yrange            The first ribbon coordinate.
  width:Float := (hi segment yrange - y0)/num  The width of a ribbon.
  for f in flist for color in 1..num repeat    For each function $f$,
    makeObject(f, xrange, yVar = y0..y0+width, create and add ribbon to
      var2Steps == 1, colorFunction == (x,y) +-> color, \_
      space == sp)                             $sp$ of a different color.
    y0 := y0 + width                           The next ribbon coordinate.
  vp := makeViewport3D(sp, "Ribbons")          Create viewport.
  drawStyle(vp, "shade")                       Select shading style.
  outlineRender(vp, "on")                      Show polygon outlines.
  showRegion(vp, "on")                         Enclose in a box.
  vp                                           Return the viewport.
```

Figure 10.2: The final **drawRibbons** function.

10.3   Coloring and Positioning Ribbons

Before leaving the ribbon example, we make two improvements. Normally, the color given to each point in the space is a function of its height within a bounding box. The points at the bottom of the box are red, those at the top are purple.

To change the normal coloring, you can give an option *colorFunction == function*. When Axiom goes about displaying the data, it determines the range of colors used for all points within the box. Axiom then distributes these numbers uniformly over the number of hues. Here we use the simple color function $(x, y) \mapsto i$ for the i-th ribbon.

Also, we add an argument *yrange* so you can give the range of y occupied by the ribbons. For example, if the *yrange* is given as $y = 0..1$ and there are 5 ribbons to be displayed, each ribbon would have width 0.2 and would appear in the range $0 \le y \le 1$.

Refer to lines 4-9. Line 4 assigns to $yVar$ the variable part of the *yrange* (after all, it need not be y). Suppose that *yrange* is given as $t = a..b$ where a and b have numerical values. Then line 5 assigns the value of a to the variable $y0$. Line 6 computes the width of the ribbon by dividing the difference of a and b by the number, *num*, of ribbons. The result is assigned to the variable *width*. Note that in the for-loop in line 7, we are iterating in parallel; it is not a nested loop.

10.4   Points, Lines, and Curves

What you have seen so far is a high-level program using the graphics facility. We
now turn to the more basic notions of points, lines, and curves in three-dimen-
sional graphs. These facilities use small floats (objects of type `DoubleFloat`)
for data. Let us first give names to the small float values 0 and 1.

The small float 0.

```
zero := 0.0@DFLOAT
```

The small float 1.

```
one := 1.0@DFLOAT
```

The `@` sign means "of the type." Thus *zero* is 0.0 of the type `DoubleFloat`. You
can also say $0.0 :: DFLOAT$.

Points can have four small float components: x, y, z coordinates and an optional
color. A "curve" is simply a list of points connected by straight line segments.

Create the point *origin* with color zero, that is, the lowest color on the color
map.

```
origin := point [zero,zero,zero,zero]
```

Create the point *unit* with color zero.

```
unit := point [one,one,one,zero]
```

Create the curve (well, here, a line) from *origin* to *unit*.

```
line := [origin, unit]
```

We make this line segment into an arrow by adding an arrowhead. The arrow-
head extends to, say, $p3$ on the left, and to, say, $p4$ on the right. To describe an
arrow, you tell Axiom to draw the two curves $[p1, p2, p3]$ and $[p2, p4]$. We also
decide through experimentation on values for *arrowScale*, the ratio of the size
of the arrowhead to the stem of the arrow, and *arrowAngle*, the angle between
the arrowhead and the arrow.

Invoke your favorite editor and create an input file called **arrows.input**. This
input file first defines the values of *arrowAngle* and *arrowScale*, then defines
the function **makeArrow**(p_1, p_2) to draw an arrow from point p_1 to p_2.

```
arrowAngle := \%pi-\%pi/10.0@DFLOAT            The angle of the arrowhead.
arrowScale := 0.2@DFLOAT                       The size of the arrowhead
                                               relative to the stem.
makeArrow(p1, p2) ==
  delta := p2 - p1                             The arrow.
  len := arrowScale * length delta             The length of the arrowhead.
  theta := atan(delta.1, delta.2)              The angle from the x-axis
  c1 := len*cos(theta + arrowAngle)            The x-coord of left endpoint
  s1 := len*sin(theta + arrowAngle)            The y-coord of left endpoint
  c2 := len*cos(theta - arrowAngle)            The x-coord of right endpoint
  s2 := len*sin(theta - arrowAngle)            The y-coord of right endpoint
  z  := p2.3*(1 - arrowScale)                  The z-coord of both endpoints
  p3 := point [p2.1 + c1, p2.2 + s1, z, p2.4]  The left endpoint of head
  p4 := point [p2.1 + c2, p2.2 + s2, z, p2.4]  The right endpoint of head
  [ [p1, p2, p3], [p2, p4] ]                   The arrow as a list of curves
```

Read the file and then create an arrow from the point *origin* to the point *unit*.

Read the input file defining **makeArrow**.

```
)read arrows
```

Construct the arrow (a list of two curves).

```
arrow := makeArrow(origin,unit)
```

Create an empty object *sp* of type *ThreeSpace*.

```
sp := createThreeSpace()
```

Add each curve of the arrow to the space *sp*.

```
for a in arrow repeat sp := curve(sp,a)
```

Create a three-dimensional viewport containing that space.

```
vp := makeViewport3D(sp,"Arrow")
```

Here is a better viewing angle.

```
rotate(vp,200,-60)
```

10.5   A Bouquet of Arrows

Let's draw a "bouquet" of arrows. Each arrow is identical. The arrowheads are uniformly placed on a circle parallel to the xy-plane. Thus the position of each arrow differs only by the angle θ, $0 \leq \theta < 2\pi$, between the arrow and the x-axis on the xy-plane.

Our bouquet is rather special: each arrow has a different color (which won't be evident here, unfortunately). This is arranged by letting the color of each successive arrow be denoted by θ. In this way, the color of arrows ranges from red to green to violet. Here is a program to draw a bouquet of n arrows.

```
drawBouquet(n,title) ==}{}
  angle := 0.0@DFLOAT                        The initial angle
  sp := createThreeSpace()                   Create empty space $sp$
  for i in 0..n-1 repeat                      For each index i, create:
    start := point [0.0@DFLOAT,0.0@DFLOAT,0.0@DFLOAT,angle]
                                              the point at base of arrow;
    end   := point [cos angle, sin angle, 1.0@DFLOAT, angle]
                                              the point at tip of arrow;
    arrow := makeArrow(start,end)            the $i$th arrow
    for a in makeArrow(start,end) repeat     For each arrow component,
      curve(sp,a)                            add the component to $sp$
    angle := angle + 2*\%pi/n                The next angle
  makeViewport3D(sp,title)                    Create the viewport from $sp$
```

Read the input file.

```
)read bouquet
```

A bouquet of a dozen arrows.

```
drawBouquet(12,"A Dozen Arrows")
```

10.6   Diversion: When Things Go Wrong

10.7   Drawing Complex Vector Fields

We now put our arrows to good use drawing complex vector fields. These vector fields give a representation of complex-valued functions of complex variables. Consider a Cartesian coordinate grid of points (x, y) in the plane, and some complex-valued function f defined on this grid. At every point on this grid, compute the value of $f(x + iy)$ and call it z. Since z has both a real and

imaginary value for a given (x, y) grid point, there are four dimensions to plot. What do we do? We represent the values of z by arrows planted at each grid point. Each arrow represents the value of z in polar coordinates (r, θ). The length of the arrow is proportional to r. Its direction is given by θ.

The code for drawing vector fields is in the file **vectors.input**. We discuss its contents from top to bottom.

Before showing you the code, we have two small matters to take care of. First, what if the function has large spikes, say, ones that go off to infinity? We define a variable *clipValue* for this purpose. When r exceeds the value of *clipValue*, then the value of *clipValue* is used instead of that for r. For convenience, we define a function *clipFun(x)* which uses *clipValue* to "clip" the value of x.

```
clipValue : DFLOAT := 6                       Maximum value allowed
clipFun(x) == min(max(x,-clipValue),clipValue)
```

Notice that we identify *clipValue* as a small float but do not declare the type of the function **clipFun**. As it turns out, **clipFun** is called with a small float value. This declaration ensures that **clipFun** never does a conversion when it is called.

The second matter concerns the possible "poles" of a function, the actual points where the spikes have infinite values. Axiom uses normal `DoubleFloat` arithmetic which does not directly handle infinite values. If your function has poles, you must adjust your step size to avoid landing directly on them (Axiom calls **error** when asked to divide a value by 0, for example).

We set the variables *realSteps* and *imagSteps* to hold the number of steps taken in the real and imaginary directions, respectively. Most examples will have ranges centered around the origin. To avoid a pole at the origin, the number of points is taken to be odd.

```
realSteps: INT := 25      Number of real steps
imagSteps: INT := 25      Number of imaginary steps
)read arrows
```

Now define the function **drawComplexVectorField** to draw the arrows. It is good practice to declare the type of the main function in the file. This one declaration is usually sufficient to ensure that other lower-level functions are compiled with the correct types.

```
C := Complex DoubleFloat
S := Segment DoubleFloat
drawComplexVectorField: (C -> C, S, S) -> VIEW3D
```

The first argument is a function mapping complex small floats into complex small floats. The second and third arguments give the range of real and imag-

inary values as segments like $a..b$. The result is a three-dimensional viewport.
Here is the full function definition:

```
drawComplexVectorField(f, realRange,imagRange) ==
  delReal := (hi(realRange)-lo(realRange))/realSteps The real step size
  delImag := (hi(imagRange)-lo(imagRange))/imagSteps The imaginary step size
  sp := createThreeSpace()                           Create empty space $sp$
  real := lo(realRange)                              The initial real value
  for i in 1..realSteps+1 repeat                     Begin real iteration
    imag := lo(imagRange)                            initial imaginary value
    for j in 1..imagSteps+1 repeat                   Begin imaginary iteration
      z := f complex(real,imag)                      value of $f$ at the point
      arg := argument z                              direction of the arrow
      len := clipFun sqrt norm z                     length of the arrow
      p1 :=  point [real, imag, 0.0@DFLOAT, arg]     base point of the arrow
      scaleLen := delReal * len                      scaled length of the arrow
      p2 := point [p1.1 + scaleLen*cos(arg),         tip point of the arrow
                   p1.2 + scaleLen*sin(arg),0.0@DFLOAT, arg]
      arrow := makeArrow(p1, p2)                     Create the arrow
      for a in arrow repeat curve(sp, a)             Add arrow to space $sp$
      imag := imag + delImag                         The next imaginary value
    real := real + delReal                           The next real value
  makeViewport3D(sp, "Complex Vector Field")         Draw it
```

As a first example, let us draw $f(z) == sin(z)$. There is no need to create a
user function: just pass the **sin** from `Complex DoubleFloat`.

Read the file.

```
)read vectors
```

Draw the complex vector field of $sin(x)$.

```
drawComplexVectorField(sin,-2..2,-2..2)
```

10.8   Drawing Complex Functions

Here is another way to graph a complex function of complex arguments. For each
complex value z, compute $f(z)$, again expressing the value in polar coordinates
(r, θ). We draw the complex valued function, again considering the (x, y)-plane
as the complex plane, using r as the height (or z-coordinate) and θ as the color.
This is a standard plot—we learned how to do this in Chapter 7 on page 297—
but here we write a new program to illustrate the creation of polygon meshes,
or grids.

Call this function **drawComplex**. It displays the points using the "mesh" of points. The function definition is in three parts.

```
drawComplex: (C -> C, S, S) -> VIEW3D
drawComplex(f, realRange, imagRange) ==                The first part
  delReal := (hi(realRange)-lo(realRange))/realSteps   The real step size
  delImag := (hi(imagRange)-lo(imagRange))/imagSteps   The imaginary step size
                                          Initial list of list of points $llp$
  llp:List List Point DFLOAT := []
```

Variables *delReal* and *delImag* give the step sizes along the real and imaginary directions as computed by the values of the global variables *realSteps* and *imagSteps*. The mesh is represented by a list of lists of points *llp*, initially empty. Now [] alone is ambiguous, so to set this initial value you have to tell Axiom what type of empty list it is. Next comes the loop which builds *llp*.

```
  real := lo(realRange)                              The initial real value
  for i in 1..realSteps+1 repeat                     Begin real iteration
    imag := lo(imagRange)                            initial imaginary value
    lp := []\$(List Point DFLOAT)                    initial list of points $lp$
    for j in 1..imagSteps+1 repeat                   Begin imaginary iteration
      z := f complex(real,imag)                      value of $f$ at the point
      pt := point [real,imag, clipFun sqrt norm z,   Create a point
                   argument z]
      lp := cons(pt,lp)                              Add the point to $lp$
      imag := imag + delImag                         The next imaginary value
    real := real + delReal                           The next real value
    llp := cons(lp, llp)                             Add $lp$ to $llp$
```

The code consists of both an inner and outer loop. Each pass through the inner loop adds one list *lp* of points to the list of lists of points *llp*. The elements of *lp* are collected in reverse order.

```
  makeViewport3D(mesh(llp), "Complex Function")    Create a mesh and display
```

The operation **mesh** then creates an object of type `ThreeSpace(DoubleFloat)` from the list of lists of points. This is then passed to **makeViewport3D** to display the image.

Now add this function directly to your **vectors.input** file and re-read the file using read vectors. We try **drawComplex** using a user-defined function f.

Read the file.

```
)read vectors
```

This one has a pole at $z = 0$.

```
f(z) == exp(1/z)
```

Draw it with an odd number of steps to avoid the pole.

```
drawComplex(f,-2..2,-2..2)
```

10.9   Functions Producing Functions

In 6.14 on page 261, you learned how to use the operation **function** to create a function from symbolic formulas. Here we introduce a similar operation which not only creates functions, but functions from functions.

The facility we need is provided by the package `MakeUnaryCompiledFunction(E,S,T)`. This package produces a unary (one-argument) compiled function from some symbolic data generated by a previous computation.[1] The E tells where the symbolic data comes from; the S and T give Axiom the source and target type of the function, respectively. The compiled function produced has type $S \to T$. To produce a compiled function with definition $p(x) == expr$, call $compiledFunction(expr, x)$ from this package. The function you get has no name. You must to assign the function to the variable p to give it that name.

Do some computation.

```
(x+1/3)**5
```

Convert this to an anonymous function of x. Assign it to the variable p to give the function a name.

```
p := compiledFunction(%,x)$MakeUnaryCompiledFunction(POLY FRAC
INT,DFLOAT,DFLOAT)
```

Apply the function.

```
p(sin(1.3))
```

For a more sophisticated application, read on.

10.10   Automatic Newton Iteration Formulas

This setting is needed to get Newton's iterations to converge.

---

[1]`MakeBinaryCompiledFunction` is available for binary functions.

```
)set streams calculate 10
```

We resume our continuing saga of arrows and complex functions. Suppose we want to investigate the behavior of Newton's iteration function in the complex plane. Given a function f, we want to find the complex values z such that $f(z) = 0$.

The first step is to produce a Newton iteration formula for a given f: $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$. We represent this formula by a function g that performs the computation on the right-hand side, that is, $x_{n+1} = g(x_n)$.

The type `Expression Integer` (abbreviated `EXPR INT`) is used to represent general symbolic expressions in Axiom. To make our facility as general as possible, we assume f has this type. Given f, we want to produce a Newton iteration function g which, given a complex point x_n, delivers the next Newton iteration point x_{n+1}.

This time we write an input file called **newton.input**. We need to import `MakeUnaryCompiledFunction` (discussed in the last section), call it with appropriate types, and then define the function *newtonStep* which references it. Here is the function *newtonStep*:

```
C := Complex DoubleFloat                        The complex numbers
complexFunPack:=MakeUnaryCompiledFunction(EXPR INT,C,C)
                                                Package for making functions

newtonStep(f) ==                                Newton's iteration function
  fun   := complexNumericFunction f             Function for $f$
  deriv := complexDerivativeFunction(f,1)       Function for $f'$
  (x:C):C +->                                   Return the iterator function
    x - fun(x)/deriv(x)

complexNumericFunction f ==                      Turn an expression $f$ into a
  v := theVariableIn f                          function
  compiledFunction(f, v)\$complexFunPack

complexDerivativeFunction(f,n) ==               Create an nth derivative
  v := theVariableIn f                          function
  df := D(f,v,n)
  compiledFunction(df, v)\$complexFunPack

theVariableIn f ==                              Returns the variable in $f$
  vl := variables f                             The list of variables
  nv := \# vl                                   The number of variables
  nv > 1 => error "Expression is not univariate."
  nv = 0 => 'x                                  Return a dummy variable
  first vl
```

Do you see what is going on here? A formula f is passed into the function **newtonStep**. First, the function turns f into a compiled program mapping complex numbers into complex numbers. Next, it does the same thing for the derivative of f. Finally, it returns a function which computes a single step of Newton's iteration.

The function **complexNumericFunction** extracts the variable from the expression f and then turns f into a function which maps complex numbers into complex numbers. The function **complexDerivativeFunction** does the same thing for the derivative of f. The function **theVariableIn** extracts the variable from the expression f, calling the function **error** if f has more than one variable. It returns the dummy variable x if f has no variables.

Let's now apply **newtonStep** to the formula for computing cube roots of two.

Read the input file with the definitions.

```
)read newton
```

```
)read vectors
```

The cube root of two.

```
f := x**3 - 2
```

Get Newton's iteration formula.

```
g := newtonStep f
```

Let a denote the result of applying Newton's iteration once to the complex number $1 + \%i$.

```
a := g(1.0 + %i)
```

Now apply it repeatedly. How fast does it converge?

```
[(a := g(a)) for i in 1..]
```

Check the accuracy of the last iterate.

```
a**3
```

In MappingPackage1, we show how functions can be manipulated as objects in Axiom. A useful operation to consider here is $*$, which means composition. For example $g * g$ causes the Newton iteration formula to be applied twice. Correspondingly, $g * *n$ means to apply the iteration formula n times.

Apply g twice to the point $1 + \%i$.

```
(g*g) (1.0 + %i)
```

Apply g 11 times.

```
(g**11) (1.0 + %i)
```

Look now at the vector field and surface generated after two steps of Newton's formula for the cube root of two. The poles in these pictures represent bad starting values, and the flat areas are the regions of convergence to the three roots.

The vector field.

```
drawComplexVectorField(g**3,-3..3,-3..3)
```

The surface.

```
drawComplex(g**3,-3..3,-3..3)
```

Chapter 11

Packages

Packages provide the bulk of Axiom's algorithmic library, from numeric packages for computing special functions to symbolic facilities for differential equations, symbolic integration, and limits.

In Chapter 10 on page 881, we developed several useful functions for drawing vector fields and complex functions. We now show you how you can add these functions to the Axiom library to make them available for general use.

The way we created the functions in Chapter 10 on page 881 is typical of how you, as an advanced Axiom user, may interact with Axiom. You have an application. You go to your editor and create an input file defining some functions for the application. Then you run the file and try the functions. Once you get them all to work, you will often want to extend them, add new features, perhaps write additional functions.

Eventually, when you have a useful set of functions for your application, you may want to add them to your local Axiom library. To do this, you embed these function definitions in a package and add that package to the library.

To introduce new packages, categories, and domains into the system, you need to use the Axiom compiler to convert the constructors into executable machine code. An existing compiler in Axiom is available on an "as-is" basis. A new, faster compiler will be available in version 2.0 of Axiom.

11.1 Names, Abbreviations, and File Structure

Each package has a name and an abbreviation. For a package of the complex draw functions from Chapter 10 on page 881, we choose the name **DrawComplex** and abbreviation **DRAWCX**.[1] To be sure that you have not chosen a name or

---

[1] An abbreviation can be any string of between two and seven capital letters and digits, beginning with a letter. See 2.2.5 on page 140 for more information.

abbreviation already used by the system, issue the system command `)show` for both the name and the abbreviation.

Once you have named the package and its abbreviation, you can choose any new filename you like with extension "**.spad**" to hold the definition of your package. We choose the name **drawpak.spad**. If your application involves more than one package, you can put them all in the same file. Axiom assumes no relationship between the name of a library file, and the name or abbreviation of a package.

Near the top of the "**.spad**" file, list all the abbreviations for the packages using `)abbrev`, each command beginning in column one. Macros giving names to Axiom expressions can also be placed near the top of the file. The macros are only usable from their point of definition until the end of the file.

Consider the definition of `DrawComplex` in Figure 11.1 on page 909. After the macro definition

```
S        ==> Segment DoubleFloat
```

the name `S` can be used in the file as a shorthand for `Segment DoubleFloat`.[2] The abbreviation command for the package

```
)abbrev package DRAWCX DrawComplex
```

is given after the macros (although it could precede them).

11.2 Syntax

The definition of a package has the syntax:

$$PackageForm : Exports \quad == \quad Implementation$$

The syntax for defining a package constructor is the same as that for defining any function in Axiom. In practice, the definition extends over many lines so that this syntax is not practical. Also, the type of a package is expressed by the operator *with* followed by an explicit list of operations. A preferable way to write the definition of a package is with a *where* expression:

The definition of a package usually has the form:
PackageForm : Exports == Implementation where
   optional type declarations
  Exports == with
    list of exported operations
  Implementation == add
    list of function definitions for exported operations

---

[2]The interpreter also allows `macro` for macro definitions.

The `DrawComplex` package takes no parameters and exports five operations, each a separate item of a *pile*. Each operation is described as a *declaration*: a name, followed by a colon (:), followed by the type of the operation. All operations have types expressed as mappings with the syntax

$$source \quad \text{->} \quad target$$

11.3 Abstract Datatypes

A constructor as defined in Axiom is called an *abstract datatype* in the computer science literature. Abstract datatypes separate "specification" (what operations are provided) from "implementation" (how the operations are implemented). The `Exports` (specification) part of a constructor is said to be "public" (it provides the user interface to the package) whereas the `Implementation` part is "private" (information here is effectively hidden—programs cannot take advantage of it).

The `Exports` part specifies what operations the package provides to users. As an author of a package, you must ensure that the `Implementation` part provides a function for each operation in the `Exports` part.[3]

An important difference between interactive programming and the use of packages is in the handling of global variables such as *realSteps* and *imagSteps*. In interactive programming, you simply change the values of variables by *assignment*. With packages, such variables are local to the package—their values can only be set using functions exported by the package. In our example package, we provide two functions **setRealSteps** and **setImagSteps** for this purpose.

Another local variable is *clipValue* which can be changed using the exported operation **setClipValue**. This value is referenced by the internal function **clipFun** that decides whether to use the computed value of the function at a point or, if the magnitude of that value is too large, the value assigned to *clipValue* (with the appropriate sign).

11.4 Capsules

The part to the right of `add` in the `Implementation` part of the definition is called a *capsule*. The purpose of a capsule is:

- to define a function for each exported operation, and

- to define a *local environment* for these functions to run.

---

[3]The `DrawComplex` package enhances the facility described in Chapter 10.8 on page 890 by allowing a complex function to have arrows emanating from the surface to indicate the direction of the complex argument.

What is a local environment? First, what is an environment? Think of the capsule as an input file that Axiom reads from top to bottom. Think of the input file as having a **)clear all** at the top so that initially no variables or functions are defined. When this file is read, variables such as *realSteps* and *arrowSize* in `DrawComplex` are set to initial values. Also, all the functions defined in the capsule are compiled. These include those that are exported (like *drawComplex*), and those that are not (like *makeArrow*). At the end, you get a set of name-value pairs: variable names (like *realSteps* and *arrowSize*) are paired with assigned values, while operation names (like *drawComplex* and *makeArrow*) are paired with function values.

This set of name-value pairs is called an *environment*. Actually, we call this environment the "initial environment" of a package: it is the environment that exists immediately after the package is first built. Afterwards, functions of this capsule can access or reset a variable in the environment. The environment is called *local* since any changes to the value of a variable in this environment can be seen *only* by these functions.

Only the functions from the package can change the variables in the local environment. When two functions are called successively from a package, any changes caused by the first function called are seen by the second.

Since the environment is local to the package, its names don't get mixed up with others in the system or your workspace. If you happen to have a variable called *realSteps* in your workspace, it does not affect what the `DrawComplex` functions do in any way.

The functions in a package are compiled into machine code. Unlike function definitions in input files that may be compiled repeatedly as you use them with varying argument types, functions in packages have a unique type (generally parameterized by the argument parameters of a package) and a unique compilation residing on disk.

The capsule itself is turned into a compiled function. This so-called *capsule function* is what builds the initial environment spoken of above. If the package has arguments (see below), then each call to the package constructor with a distinct pair of arguments builds a distinct package, each with its own local environment.

11.5   Input Files vs. Packages

A good question at this point would be "Is writing a package more difficult than writing an input file?"

The programs in input files are designed for flexibility and ease-of-use. Axiom can usually work out all of your types as it reads your program and does the computations you request. Let's say that you define a one-argument function without giving its type. When you first apply the function to a value, this value

is understood by Axiom as identifying the type for the argument parameter. Most of the time Axiom goes through the body of your function and figures out the target type that you have in mind. Axiom sometimes fails to get it right. Then—and only then—do you need a declaration to tell Axiom what type you want.

Input files are usually written to be read by Axiom—and by you. Without suitable documentation and declarations, your input files are likely incomprehensible to a colleague—and to you some months later!

Packages are designed for legibility, as well as run-time efficiency. There are few new concepts you need to learn to write packages. Rather, you just have to be explicit about types and type conversions. The types of all functions are pre-declared so that Axiom—and the reader— knows precisely what types of arguments can be passed to and from the functions (certainly you don't want a colleague to guess or to have to work this out from context!). The types of local variables are also declared. Type conversions are explicit, never automatic.[4]

In summary, packages are more tedious to write than input files. When writing input files, you can casually go ahead, giving some facts now, leaving others for later. Writing packages requires forethought, care and discipline.

11.6 Compiling Packages

Once you have defined the package `DrawComplex`, you need to compile and test it. To compile the package, issue the system command `)compile drawpak`. Axiom reads the file **drawpak.spad** and compiles its contents into machine binary. If all goes well, the file `DRAWCX.NRLIB` is created in your local directory for the package. To test the package, you must load the package before trying an operation.

Compile the package.

)compile drawpak

Expose the package.

)expose DRAWCX

Use an odd step size to avoid a pole at the origin.

setRealSteps 51

setImagSteps 51

---

[4]There is one exception to this rule: conversions from a subdomain to a domain are automatic. After all, the objects both have the domain as a common type.

Define **f** to be the Gamma function.

```
f(z) == Gamma(z)
```

Clip values of function with magnitude larger than 7.

```
setClipValue 7
```

Draw the **Gamma** function.

```
drawComplex(f,-%pi..%pi,-%pi..%pi, false)
```

11.7 Parameters

The power of packages becomes evident when packages have parameters. Usually these parameters are domains and the exported operations have types involving these parameters.

In Chapter 2 on page 129, you learned that categories denote classes of domains. Although we cover this notion in detail in the next chapter, we now give you a sneak preview of its usefulness.

In 6.15 on page 265, we defined functions $bubbleSort(m)$ and $insertionSort(m)$ to sort a list of integers. If you look at the code for these functions, you see that they may be used to sort *any* structure m with the right properties. Also, the functions can be used to sort lists of *any* elements—not just integers. Let us now recall the code for *bubbleSort*.

```
bubbleSort(m) ==
  n := #m
  for i in 1..(n-1) repeat
    for j in n..(i+1) by -1 repeat
      if m.j < m.(j-1) then swap!(m,j,j-1)
  m
```

What properties of "lists of integers" are assumed by the sorting algorithm? In the first line, the operation **#** computes the maximum index of the list. The first obvious property is that m must have a finite number of elements. In Axiom, this is done by your telling Axiom that m has the "attribute" **finiteAggregate**. An *attribute* is a property that a domain either has or does not have. As we show later in 12.9 on page 918, programs can query domains as to the presence or absence of an attribute.

The operation **swap** swaps elements of m. Using Browse, you find that **swap** requires its elements to come from a domain of category `IndexedAggregate` with

attribute `shallowlyMutable`. This attribute means that you can change the internal components of m without changing its external structure. Shallowly-mutable data structures include lists, streams, one- and two-dimensional arrays, vectors, and matrices.

The category `IndexedAggregate` designates the class of aggregates whose elements can be accessed by the notation $m.s$ for suitable selectors s. The category `IndexedAggregate` takes two arguments: *Index*, a domain of selectors for the aggregate, and *Entry*, a domain of entries for the aggregate. Since the sort functions access elements by integers, we must choose *Index* =`Integer`. The most general class of domains for which *bubbleSort* and *insertionSort* are defined are those of category `IndexedAggregate(Integer,Entry)` with the two attributes **shallowlyMutable** and **finiteAggregate**.

Using Browse, you can also discover that Axiom has many kinds of domains with attribute **shallowlyMutable**. Those of class `IndexedAggregate(Integer,Entry)` include `Bits`, `FlexibleArray`, `OneDimensionalArray`, `List`, `String`, and `Vector`, and also `HashTable` and `EqTable` with integer keys. Although you may never want to sort all such structures, we nonetheless demonstrate Axiom's ability to do so.

Another requirement is that `Entry` has an operation `<`. One way to get this operation is to assume that `Entry` has category `OrderedSet`. By definition, will then export a `<` operation. A more general approach is to allow any comparison function f to be used for sorting. This function will be passed as an argument to the sorting functions.

Our sorting package then takes two arguments: a domain S of objects of *any* type, and a domain A, an aggregate of type `IndexedAggregate(Integer, S)` with the above two attributes. Here is its definition using what are close to the original definitions of *bubbleSort* and *insertionSort* for sorting lists of integers. The symbol `!` is added to the ends of the operation names. This uniform naming convention is used for Axiom operation names that destructively change one or more of their arguments.

```
SortPackage(S,A) : Exports == Implementation where
  S: Object
  A: IndexedAggregate(Integer,S)
    with (finiteAggregate; shallowlyMutable)

  Exports == with
    bubbleSort!: (A,(S,S) -> Boolean) -> A
    insertionSort!: (A, (S,S) -> Boolean) -> A

  Implementation == add
    bubbleSort!(m,f) ==
      n := \#m
      for i in 1..(n-1) repeat
```

```
        for j in n..(i+1) by -1 repeat
          if f(m.j,m.(j-1)) then swap!(m,j,j-1)
    m
  insertionSort!(m,f) ==
    for i in 2..\#m repeat
      j := i
      while j > 1 and f(m.j,m.(j-1)) repeat
        swap!(m,j,j-1)
        j := (j - 1) pretend PositiveInteger
    m
```

11.8   Conditionals

When packages have parameters, you can say that an operation is or is not
exported depending on the values of those parameters. When the domain of
objects S has an < operation, we can supply one-argument versions of *bubbleSort*
and *insertionSort* which use this operation for sorting. The presence of the
operation < is guaranteed when S is an ordered set.

```
Exports == with
    bubbleSort!: (A,(S,S) -> Boolean) -> A
    insertionSort!: (A, (S,S) -> Boolean) -> A

    if S has OrderedSet then
      bubbleSort!: A -> A
      insertionSort!: A -> A
```

In addition to exporting the one-argument sort operations conditionally, we
must provide conditional definitions for the operations in the `Implementation`
part. This is easy: just have the one-argument functions call the corresponding
two-argument functions with the operation < from S.

```
  Implementation == add
      ...
    if S has OrderedSet then
      bubbleSort!(m) == bubbleSort!(m,<\$S)
      insertionSort!(m) == insertionSort!(m,<\$S)
```

In 6.15 on page 265, we give an alternative definition of **bubbleSort** using **first**
and **rest** that is more efficient for a list (for which access to any element requires
traversing the list from its first node). To implement a more efficient algorithm
for lists, we need the operation **setelt** which allows us to destructively change
the **first** and **rest** of a list. Using Browse, you find that these operations come
from category `UnaryRecursiveAggregate`. Several aggregate types are unary

recursive aggregates including those of `List` and `AssociationList`. We provide two different implementations for **bubbleSort!** and **insertionSort!**: one for list-like structures, another for array-like structures.

```
Implementation == add
      ...
   if A has UnaryRecursiveAggregate(S) then
     bubbleSort!(m,fn) ==
       empty? m => m
       l := m
       while not empty? (r := l.rest) repeat
          r := bubbleSort! r
          x := l.first
          if fn(r.first,x) then
            l.first := r.first
            r.first := x
          l.rest := r
          l := l.rest
        m
     insertionSort!(m,fn) ==
        ...
```

The ordering of definitions is important. The standard definitions come first and then the predicate

```
A has UnaryRecursiveAggregate(S)
```

is evaluated. If `true`, the special definitions cover up the standard ones.

Another equivalent way to write the capsule is to use an $if - then - else$ expression:

```
   if A has UnaryRecursiveAggregate(S) then
      ...
   else
      ...
```

11.9   Testing

Once you have written the package, embed it in a file, for example, **sortpak.spad**. Be sure to include an **)abbrev** command at the top of the file:

```
)abbrev package SORTPAK SortPackage
```

Now compile the file (using **)compile sortpak.spad**).

Expose the constructor. You are then ready to begin testing.

```
)expose SORTPAK
```

Define a list.

```
l := [1,7,4,2,11,-7,3,2]
```

Since the integers are an ordered set, a one-argument operation will do.

```
bubbleSort!(l)
```

Re-sort it using "greater than."

```
bubbleSort!(l,(x,y) +-> x > y)
```

Now sort it again using < on integers.

```
bubbleSort!(l, <$Integer)
```

A string is an aggregate of characters so we can sort them as well.

```
bubbleSort!  "Mathematical Sciences"
```

Is < defined on booleans?

```
false < true
```

Good! Create a bit string representing ten consecutive boolean values `true`.

```
u :  Bits := new(10,true)
```

Set bits 3 through 5 to `false`, then display the result.

```
u(3..5) := false; u
```

Now sort these booleans.

```
bubbleSort!  u
```

Create an "eq-table", a table having integers as keys and strings as values.

```
t :  EqTable(Integer,String) := table()
```

Give the table a first entry.

```
t.1 := "robert"
```

And a second.

```
t.2 := "richard"
```

What does the table look like?

```
t
```

Now sort it.

```
bubbleSort!  t
```

11.10 How Packages Work

Recall that packages as abstract datatypes are compiled independently and put into the library. The curious reader may ask: "How is the interpreter able to find an operation such as **bubbleSort!**? Also, how is a single compiled function such as **bubbleSort!** able to sort data of different types?"

After the interpreter loads the package `SortPackage`, the four operations from the package become known to the interpreter. Each of these operations is expressed as a *modemap* in which the type of the operation is written in terms of symbolic domains.

See the modemaps for **bubbleSort!**.

)display op bubbleSort!

```
There are 2 exposed functions called bubbleSort! :

   [1] D1 -> D1 from SortPackage(D2,D1)
         if D2 has ORDSET and D2 has OBJECT and D1 has
         IndexedAggregate(Integer, D2) with
             finiteAggregate
             shallowlyMutable

   [2] (D1,((D3,D3) -> Boolean)) -> D1 from SortPackage(D3,D1)
         if D3 has OBJECT and D1 has
         IndexedAggregate(Integer,D3) with
             finiteAggregate
             shallowlyMutable
```

What happens if you ask for *bubbleSort*!($[1, -5, 3]$)? There is a unique modemap for an operation named **bubbleSort!** with one argument. Since $[1, -5, 3]$ is a list of integers, the symbolic domain $D1$ is defined as `List(Integer)`. For some operation to apply, it must satisfy the predicate for some $D2$. What $D2$? The third expression of the `and` requires `D1 has IndexedAggregate(Integer, D2) with` two attributes. So the interpreter searches for an `IndexedAggregate` among the ancestors of `List (Integer)` (see 12.4 on page 914). It finds one: `IndexedAggregate(Integer, Integer)`. The interpreter tries defining $D2$ as `Integer`. After substituting for $D1$ and $D2$, the predicate evaluates to `true`. An applicable operation has been found!

Now Axiom builds the package `SortPackage(List(Integer), Integer)`. According to its definition, this package exports the required operation: **bubbleSort!**: List Integer → List Integer. The interpreter then asks the package for a function implementing this operation. The package gets all the functions it needs (for example, **rest** and **swap**) from the appropriate domains and then it returns a **bubbleSort!** to the interpreter together with the local environment for **bubbleSort!**. The interpreter applies the function to the argument $[1, -5, 3]$. The **bubbleSort!** function is executed in its local environment and produces the result.

```
C      ==> Complex DoubleFloat              All constructors used in a file
S      ==> Segment DoubleFloat              must be spelled out in full
INT    ==> Integer                          unless abbreviated by macros
DFLOAT ==> DoubleFloat                      like these at the top of
VIEW3D ==> ThreeDimensionalViewport         a file
CURVE  ==> List List Point DFLOAT

)abbrev package DRAWCX DrawComplex          Identify kinds and abbreviations
DrawComplex(): Exports == Implementation where Type definition begins here

  Exports == with                                   Export part begins
    drawComplex: (C -> C,S,S,Boolean) -> VIEW3D        Exported Operations
    drawComplexVectorField: (C -> C,S,S) -> VIEW3D
    setRealSteps: INT -> INT
    setImagSteps: INT -> INT
    setClipValue: DFLOAT-> DFLOAT

  Implementation == add                          Implementation part begins
    arrowScale : DFLOAT := (0.2)::DFLOAT --relative size Local variable 1
    arrowAngle : DFLOAT := pi()-pi()/(20::DFLOAT)       Local variable 2
    realSteps  : INT := 11 --\# real steps             Local variable 3
    imagSteps  : INT := 11 --\# imaginary steps         Local variable 4
    clipValue  : DFLOAT  := 10::DFLOAT --maximum vector length
                                                        Local variable 5

    setRealSteps(n) == realSteps := n        Exported function definition 1
    setImagSteps(n) == imagSteps := n        Exported function definition 2
    setClipValue(c) == clipValue := c        Exported function definition 3

    clipFun: DFLOAT -> DFLOAT --Clip large magnitudes.
    clipFun(x) == min(max(x, -clipValue), clipValue)
                                             Local function definition 1

    makeArrow: (Point DFLOAT,Point DFLOAT,DFLOAT,DFLOAT) -> CURVE
    makeArrow(p1, p2, len, arg) == ...       Local function definition 2

    drawComplex(f, realRange, imagRange, arrows?) == ...
                                             Exported function definition 4
```

Figure 11.1: The DrawComplex package.

Chapter 12

Categories

This chapter unravels the mysteries of categories—what they are, how they are related to domains and packages, how they are defined in Axiom, and how you can extend the system to include new categories of your own.

We assume that you have read the introductory material on domains and categories in 2.1.1 on page 131. There you learned that the notion of packages covered in the previous chapter are special cases of domains. While this is in fact the case, it is useful here to regard domains as distinct from packages.

Think of a domain as a datatype, a collection of objects (the objects of the domain). From your "sneak preview" in the previous chapter, you might conclude that categories are simply named clusters of operations exported by domains. As it turns out, categories have a much deeper meaning. Categories are fundamental to the design of Axiom. They control the interactions between domains and algorithmic packages, and, in fact, between all the components of Axiom.

Categories form hierarchies as shown on the inside cover pages of this book. The inside front-cover pages illustrate the basic algebraic hierarchy of the Axiom programming language. The inside back-cover pages show the hierarchy for data structures.

Think of the category structures of Axiom as a foundation for a city on which superstructures (domains) are built. The algebraic hierarchy, for example, serves as a foundation for constructive mathematical algorithms embedded in the domains of Axiom. Once in place, domains can be constructed, either independently or from one another.

Superstructures are built for quality—domains are compiled into machine code for run-time efficiency. You can extend the foundation in directions beyond the space directly beneath the superstructures, then extend selected superstructures to cover the space. Because of the compilation strategy, changing components of the foundation generally means that the existing superstructures (domains) built on the changed parts of the foundation (categories) have to be rebuilt—

that is, recompiled.

Before delving into some of the interesting facts about categories, let's see how you define them in Axiom.

12.1   Definitions

A category is defined by a function with exactly the same format as any other function in Axiom.

The definition of a category has the syntax:

CategoryForm : `Category`   ==   *Extensions* [`with` *Exports*]

The brackets [ ] here indicate optionality.

The first example of a category definition is `SetCategory`, the most basic of the algebraic categories in Axiom.

```
SetCategory(): Category ==
   Join(Type,CoercibleTo OutputForm) with
      "=" : (\$, \$) -> Boolean
```

The definition starts off with the name of the category (`SetCategory`); this is always in column one in the source file. All parts of a category definition are then indented with respect to this first line.

In Chapter 2 on page 129, we talked about `Ring` as denoting the class of all domains that are rings, in short, the class of all rings. While this is the usual naming convention in Axiom, it is also common to use the word "Category" at the end of a category name for clarity. The interpretation of the name `SetCategory` is, then, "the category of all domains that are (mathematical) sets."

The name `SetCategory` is followed in the definition by its formal parameters enclosed in parentheses (). Here there are no parameters. As required, the type of the result of this category function is the distinguished name `Category`.

Then comes the ==. As usual, what appears to the right of the == is a definition, here, a category definition. A category definition always has two parts separated by the reserved word *with*.

The first part tells what categories the category extends. Here, the category extends two categories: `Type`, the category of all domains, and `CoercibleTo(OutputForm)`. The operation *Join* is a system-defined operation that forms a single category from two or more other categories.

Every category other than `Type` is an extension of some other category. If, for example, `SetCategory` extended only the category `Type`, the definition here

would read "`Type with ...`". In fact, the `Type` is optional in this line; "`with ...`" suffices.

12.2 Exports

To the right of the *with* is a list of all the exports of the category. Each exported operation has a name and a type expressed by a *declaration* of the form "*name*: *type*".

Categories can export symbols, as well as `0` and `1` which denote domain constants.[1] In the current implementation, all other exports are operations with types expressed as mappings with the syntax

$$source \quad \text{->} \quad target$$

The category `SetCategory` has a single export: the operation = whose type is given by the mapping `($, $) -> Boolean`. The `$` in a mapping type always means "the domain." Thus the operation = takes two arguments from the domain and returns a value of type `Boolean`.

The source part of the mapping here is given by a *tuple* consisting of two or more types separated by commas and enclosed in parentheses. If an operation takes only one argument, you can drop the parentheses around the source type. If the mapping has no arguments, the source part of the mapping is either left blank or written as `()`. Here are examples of formats of various operations with some contrived names.

```
someIntegerConstant  :    $
aZeroArgumentOperation:   () -> Integer
aOneArgumentOperation:    Integer -> $
aTwoArgumentOperation:    (Integer,$) -> Void
aThreeArgumentOperation:  ($,Integer,$) -> Fraction($)
```

12.3 Documentation

The definition of `SetCategory` above is missing an important component: its library documentation. Here is its definition, complete with documentation.

```
++ Description:
++ \bs{}axiomType\{SetCategory\} is the basic category
++ for describing a collection of elements with
++ \bs{}axiomOp\{=\} (equality) and a \bs{}axiomFun\{coerce\}
++ to \bs{}axiomType\{OutputForm\}.
```

---

[1]The numbers `0` and `1` are operation names in Axiom.

```
SetCategory(): Category ==
  Join(Type, CoercibleTo OutputForm) with
    "=": (\$, \$) -> Boolean
      ++ \bs{}axiom\{x = y\} tests if \bs{}axiom\{x\} and
      ++ \bs{}axiom\{y\} are equal.
```

Documentary comments are an important part of constructor definitions. Documentation is given both for the category itself and for each export. A description for the category precedes the code. Each line of the description begins in column one with `++`. The description starts with the word `Description:`[2] All lines of the description following the initial line are indented by the same amount.

Surround the name of any constructor (with or without parameters) with an `{\bf }`. Similarly, surround an operator name with `{\tt }`, an Axiom operation with `{\bf }`, and a variable or Axiom expression with `$$`. Library documentation is given in a TeX-like language so that it can be used both for hard-copy and for Browse. These different wrappings cause operations and types to have mouse-active buttons in Browse. For hard-copy output, wrapped expressions appear in a different font. The above documentation appears in hard-copy as:

> `SetCategory` is the basic category for describing a collection of elements with `=` (equality) and a **coerce** to `OutputForm`.

and

> $x = y$ tests if x and y are equal.

For our purposes in this chapter, we omit the documentation from further category descriptions.

12.4   Hierarchies

A second example of a category is `SemiGroup`, defined by:

```
SemiGroup(): Category == SetCategory with
    "*":  (\$,\$) -> \$
    "**": (\$, PositiveInteger) -> \$
```

This definition is as simple as that for `SetCategory`, except that there are two exported operations. Multiple exported operations are written as a *pile*, that is, they all begin in the same column. Here you see that the category mentions

---

[2]Other information such as the author's name, date of creation, and so on, can go in this area as well but are currently ignored by Axiom.

another type, `PositiveInteger`, in a signature. Any domain can be used in a signature.

Since categories extend one another, they form hierarchies. Each category other than `Type` has one or more parents given by the one or more categories mentioned before the *with* part of the definition. `SemiGroup` extends `SetCategory` and `SetCategory` extends both `Type` and `CoercibleTo (OutputForm)`. Since `CoercibleTo (OutputForm)` also extends `Type`, the mention of `Type` in the definition is unnecessary but included for emphasis.

12.5 Membership

We say a category designates a class of domains. What class of domains? That is, how does Axiom know what domains belong to what categories? The simple answer to this basic question is key to the design of Axiom:

Domains belong to categories by assertion.

When a domain is defined, it is asserted to belong to one or more categories. Suppose, for example, that an author of domain `String` wishes to use the binary operator $*$ to denote concatenation. Thus "*hello*" $*$ "*there*" would produce the string "*hellothere*"[3]. The author of `String` could then assert that `String` is a member of `SemiGroup`. According to our definition of `SemiGroup`, strings would then also have the operation $**$ defined automatically. Then " $--$" $**4$ would produce a string of eight dashes " $--------$". Since `String` is a member of `SemiGroup`, it also is a member of `SetCategory` and thus has an operation $=$ for testing that two strings are equal.

Now turn to the algebraic category hierarchy inside the front cover of this book. Any domain that is a member of a category extending `SemiGroup` is a member of `SemiGroup` (that is, it *is* a semigroup). In particular, any domain asserted to be a `Ring` is a semigroup since `Ring` extends `Monoid`, that, in turn, extends `SemiGroup`. The definition of `Integer` in Axiom asserts that `Integer` is a member of category `IntegerNumberSystem`, that, in turn, asserts that it is a member of `EuclideanDomain`. Now `EuclideanDomain` extends `PrincipalIdealDomain` and so on. If you trace up the hierarchy, you see that `EuclideanDomain` extends `Ring`, and, therefore, `SemiGroup`. Thus `Integer` is a semigroup and also exports the operations $*$ and $**$.

12.6 Defaults

We actually omitted the last part of the definition of `SemiGroup` in 12.4 on page 914. Here now is its complete Axiom definition.

---

[3]Actually, concatenation of strings in Axiom is done by juxtaposition or by using the operation **concat**. The expression "*hello*""*there*" produces the string "*hellothere*".

```
SemiGroup(): Category == SetCategory with
     "*": (\$, \$) -> \$
     "**": (\$, PositiveInteger) -> \$
    add
      import RepeatedSquaring(\$)
      x: \$ ** n: PositiveInteger == expt(x,n)
```

The *add* part at the end is used to give "default definitions" for exported operations. Once you have a multiplication operation $*$, you can define exponentiation for positive integer exponents using repeated multiplication:

$$x^n = \underbrace{x\,x\,x\,\cdots\,x}_{n\ \text{times}}$$

This definition for $**$ is called a *default* definition. In general, a category can give default definitions for any operation it exports. Since `SemiGroup` and all its category descendants in the hierarchy export $**$, any descendant category may redefine $**$ as well.

A domain of category `SemiGroup` (such as `Integer`) may or may not choose to define its own $**$ operation. If it does not, a default definition that is closest (in a "tree-distance" sense of the hierarchy) to the domain is chosen.

The part of the category definition following an *add* operation is a *capsule*, as discussed in the previous chapter. The line

```
import RepeatedSquaring($)
```

references the package `RepeatedSquaring($)`, that is, the package `RepeatedSquaring` that takes "this domain" as its parameter. For example, if the semigroup `Polynomial (Integer)` does not define its own exponentiation operation, the definition used may come from the package `RepeatedSquaring (Polynomial (Integer))`. The next line gives the definition in terms of **expt** from that package.

The default definitions are collected to form a "default package" for the category. The name of the package is the same as the category but with an ampersand (`&`) added at the end. A default package always takes an additional argument relative to the category. Here is the definition of the default package `SemiGroup&` as automatically generated by Axiom from the above definition of `SemiGroup`.

```
SemiGroup\_\&(\$): Exports == Implementation where
  \$: SemiGroup
  Exports == with
    "**": (\$, PositiveInteger) -> \$
  Implementation == add
    import RepeatedSquaring(\$)
    x:\$ ** n:PositiveInteger == expt(x,n)
```

12.7 Axioms

In the previous section you saw the complete Axiom program defining `SemiGroup`. According to this definition, semigroups (that is, are sets with the operations "`*`" and "`**`".

You might ask: "Aside from the notion of default packages, isn't a category just a *macro*, that is, a shorthand equivalent to the two operations $*$ and $**$ with their types?" If a category were a macro, every time you saw the word `SemiGroup`, you would rewrite it by its list of exported operations. Furthermore, every time you saw the exported operations of `SemiGroup` among the exports of a constructor, you could conclude that the constructor exported `SemiGroup`.

A category is *not* a macro and here is why. The definition for `SemiGroup` has documentation that states:

> Category `SemiGroup` denotes the class of all multiplicative semigroups, that is, a set with an associative operation $*$.
>
> Axioms:
> `associative("*" :  ($,$)->$) -- (x*y)*z = x*(y*z)`

According to the author's remarks, the mere exporting of an operation named $*$ and $**$ is not enough to qualify the domain as a `SemiGroup`. In fact, a domain can be a semigroup only if it explicitly exports a $**$ and a $*$ satisfying the associativity axiom.

In general, a category name implies a set of axioms, even mathematical theorems. There are numerous axioms from `Ring`, for example, that are well-understood from the literature. No attempt is made to list them all. Nonetheless, all such mathematical facts are implicit by the use of the name `Ring`.

12.8 Correctness

While such statements are only comments, Axiom can enforce their intention simply by shifting the burden of responsibility onto the author of a domain. A domain belongs to category *Ring* only if the author asserts that the domain belongs to `Ring` or to a category that extends `Ring`.

This principle of assertion is important for large user-extendable systems. Axiom has a large library of operations offering facilities in many areas. Names such as **norm** and **product**, for example, have diverse meanings in diverse contexts. An inescapable hindrance to users would be to force those who wish to extend Axiom to always invent new names for operations. Axiom allows you to reuse names, and then use context to disambiguate one from another.

Here is another example of why this is important. Some languages, such as **APL**, denote the `Boolean` constants `true` and `false` by the integers 1 and 0.

You may want to let infix operators $+$ and $*$ serve as the logical operators **or**
and **and**, respectively. But note this: `Boolean` is not a ring. The *inverse axiom*
for `Ring` states:

> Every element x has an additive inverse y such that $x + y = 0$.

`Boolean` is not a ring since `true` has no inverse—there is no inverse element a
such that $1 + a = 0$ (in terms of booleans, `(true or a) = false`). Nonethe-
less, Axiom *could* easily and correctly implement `Boolean` this way. `Boolean`
simply would not assert that it is of category `Ring`. Thus the "+" for `Boolean`
values is not confused with the one for `Ring`. Since the `Polynomial` construc-
tor requires its argument to be a ring, Axiom would then refuse to build the
domain `Polynomial(Boolean)`. Also, Axiom would refuse to wrongfully apply
algorithms to `Boolean` elements that presume that the ring axioms for "+" hold.

12.9   Attributes

Most axioms are not computationally useful. Those that are can be explicitly
expressed by what Axiom calls an *attribute*. The attribute **commutative("*")**,
for example, is used to assert that a domain has commutative multiplication.
Its definition is given by its documentation:

> A domain R has **commutative("*")** if it has an operation "*": $(R,R) \rightarrow$
> R such that $x * y = y * x$.

Just as you can test whether a domain has the category `Ring`, you can test that
a domain has a given attribute.

Do polynomials over the integers have commutative multiplication?

```
Polynomial Integer has commutative("*")
```

Do matrices over the integers have commutative multiplication?

```
Matrix Integer has commutative("*")
```

Attributes are used to conditionally export and define operations for a domain
(see). Attributes can also be asserted in a category definition.

After mentioning category `Ring` many times in this book, it is high time that
we show you its definition:

```
Ring(): Category ==
  Join(Rng,Monoid,LeftModule(\$: Rng)) with
      characteristic: -> NonNegativeInteger
      coerce: Integer -> \$
```

```
        unitsKnown
    add
      n:Integer
      coerce(n) == n * 1\$\$
```

There are only two new things here. First, look at the `$$` on the last line. This is not a typographic error! The first `$` says that the 1 is to come from some domain. The second `$` says that the domain is "this domain." If `$` is `Fraction(Integer)`, this line reads `coerce(n) == n * 1$Fraction(Integer)`.

The second new thing is the presence of attribute *"unitsKnown"*. Axiom can always distinguish an attribute from an operation. An operation has a name and a type. An attribute has no type. The attribute **unitsKnown** asserts a rather subtle mathematical fact that is normally taken for granted when working with rings.[4] Because programs can test for this attribute, Axiom can correctly handle rather more complicated mathematical structures (ones that are similar to rings but do not have this attribute).

12.10 Parameters

Like domain constructors, category constructors can also have parameters. For example, category `MatrixCategory` is a parameterized category for defining matrices over a ring R so that the matrix domains can have different representations and indexing schemes. Its definition has the form:

```
MatrixCategory(R,Row,Col): Category ==
    TwoDimensionalArrayCategory(R,Row,Col) with ...
```

The category extends `TwoDimensionalArrayCategory` with the same arguments. You cannot find `TwoDimensionalArrayCategory` in the algebraic hierarchy listing. Rather, it is a member of the data structure hierarchy, given inside the back cover of this book. In particular, `TwoDimensionalArrayCategory` is an extension of `HomogeneousAggregate` since its elements are all one type.

The domain `Matrix(R)`, the class of matrices with coefficients from domain R, asserts that it is a member of category `MatrixCategory(R, Vector(R), Vector(R))`. The parameters of a category must also have types. The first parameter to `MatrixCategory` R is required to be a ring. The second and third are required to be domains of category `FiniteLinearAggregate(R)`.[5] In practice, examples of categories having parameters other than domains are rare.

---

[4]With this axiom, the units of a domain are the set of elements x that each have a multiplicative inverse y in the domain. Thus 1 and -1 are units in domain `Integer`. Also, for `Fraction Integer`, the domain of rational numbers, all non-zero elements are units.

[5]This is another extension of `HomogeneousAggregate` that you can see in the data structure hierarchy.

Adding the declarations for parameters to the definition for `MatrixCategory`, we have:

```
R: Ring
(Row, Col): FiniteLinearAggregate(R)

MatrixCategory(R, Row, Col): Category ==
    TwoDimensionalArrayCategory(R, Row, Col) with ...
```

12.11   Conditionals

As categories have parameters, the actual operations exported by a category can depend on these parameters. As an example, the operation **determinant** from category `MatrixCategory` is only exported when the underlying domain R has commutative multiplication:

```
if R has commutative("*") then
   determinant: $ -> R
```

Conditionals can also define conditional extensions of a category. Here is a portion of the definition of `QuotientFieldCategory`:

```
QuotientFieldCategory(R) : Category == ... with ...
     if R has OrderedSet then OrderedSet
     if R has IntegerNumberSystem then
       ceiling: \$ -> R
         ...
```

Think of category `QuotientFieldCategory(R)` as denoting the domain `Fraction(R)`, the class of all fractions of the form a/b for elements of R. The first conditional means in English: "If the elements of R are totally ordered (R is an `OrderedSet`), then so are the fractions a/b".

The second conditional is used to conditionally export an operation **ceiling** which returns the smallest integer greater than or equal to its argument. Clearly, "ceiling" makes sense for integers but not for polynomials and other algebraic structures. Because of this conditional, the domain `Fraction(Integer)` exports an operation **ceiling**: Fraction Integer $\rightarrow$ Integer, but `Fraction Polynomial Integer` does not.

Conditionals can also appear in the default definitions for the operations of a category. For example, a default definition for **ceiling** within the part following the *add* reads:

```
if R has IntegerNumberSystem then
    ceiling x == ...
```

Here the predicate used is identical to the predicate in the `Exports` part. This need not be the case. See 11.8 on page 904 for a more complicated example.

12.12   Anonymous Categories

The part of a category to the right of a `with` is also regarded as a category—an "anonymous category." Thus you have already seen a category definition in Chapter 11 on page 897. The `Exports` part of the package `DrawComplex` (11.3 on page 899) is an anonymous category. This is not necessary. We could, instead, give this category a name:

```
DrawComplexCategory(): Category == with
   drawComplex: (C -> C,S,S,Boolean) -> VIEW3D
   drawComplexVectorField: (C -> C,S,S) -> VIEW3D
   setRealSteps: INT -> INT
   setImagSteps: INT -> INT
   setClipValue: DFLOAT-> DFLOAT
```

and then define `DrawComplex` by:

```
DrawComplex(): DrawComplexCategory == Implementation
   where
      ...
```

There is no reason, however, to give this list of exports a name since no other domain or package exports it. In fact, it is rare for a package to export a named category. As you will see in the next chapter, however, it is very common for the definition of domains to mention one or more category before the `with`.

```
)read alql.boot
```

```
)load DLIST ICARD DBASE QEQUAT MTHING OPQUERY )update
```

Chapter 13

Domains

We finally come to the *domain constructor*. A few subtle differences between packages and domains turn up some interesting issues. We first discuss these differences then describe the resulting issues by illustrating a program for the `QuadraticForm` constructor. After a short example of an algebraic constructor, `CliffordAlgebra`, we show how you use domain constructors to build a database query facility.

13.1   Domains vs. Packages

Packages are special cases of domains. What is the difference between a package and a domain that is not a package? By definition, there is only one difference: a domain that is not a package has the symbol `$` appearing somewhere among the types of its exported operations. The `$` denotes "this domain." If the `$` appears before the `->` in the type of a signature, it means the operation takes an element from the domain as an argument. If it appears after the `->`, then the operation returns an element of the domain.

If no exported operations mention `$`, then evidently there is nothing of interest to do with the objects of the domain. You might then say that a package is a "boring" domain! But, as you saw in Chapter 11 on page 897, packages are a very useful notion indeed. The exported operations of a package depend solely on the parameters to the package constructor and other explicit domains.

To summarize, domain constructors are versatile structures that serve two distinct practical purposes: Those like `Polynomial` and `List` describe classes of computational objects; others, like `SortPackage`, describe packages of useful operations. As in the last chapter, we focus here on the first kind.

13.2   Definitions

The syntax for defining a domain constructor is the same as for any function in
Axiom:

$$DomainForm \ : \ Exports \ \texttt{==} \ Implementation$$

As this definition usually extends over many lines, a *where* expression is gener-
ally used instead.

A recommended format for the definition of a domain is:
DomainForm :   Exports == Implementation where
    optional type declarations
    Exports == [*Category Assertions*] with
        list of exported operations
    Implementation == [*Add Domain*] add
        [Rep := *Representation*]
        list of function definitions for exported operations

Note: The brackets [ ] here denote optionality.

A complete domain constructor definition for `QuadraticForm` is shown in Figure
13.1 on page 925. Interestingly, this little domain illustrates all the new concepts
you need to learn.

A domain constructor can take any number and type of parameters.
`QuadraticForm` takes a positive integer n and a field K as arguments. Like
a package, a domain has a set of explicit exports and an implementation de-
scribed by a capsule. Domain constructors are documented in the same way as
package constructors.

Domain `QuadraticForm(n, K)`, for a given positive integer n and domain K,
explicitly exports three operations:

- *quadraticForm*(A) creates a quadratic form from a matrix A.

- *matrix*(q) returns the matrix A used to create the quadratic form q.

- $q.v$ computes the scalar $v^T A v$ for a given vector v.

Compared with the corresponding syntax given for the definition of a package,
you see that a domain constructor has three optional parts to its definition:
Category Assertions, *Add Domain*, and *Representation*.

13.3   Category Assertions

The *Category Assertions* part of your domain constructor definition lists those
categories of which all domains created by the constructor are unconditionally

```
)abbrev domain QFORM QuadraticForm

++ Description:
++   This domain provides modest support for
++   quadratic forms.
QuadraticForm(n, K): Exports == Implementation where
    n: PositiveInteger
    K: Field

    Exports == AbelianGroup with            --The exports
      quadraticForm: SquareMatrix(n,K) -> $ --export this
        ++ \bs{}axiom\{quadraticForm(m)\} creates a quadratic
        ++ quadratic form from a symmetric,
        ++ square matrix \bs{}axiom\{m\}.
      matrix: \$ -> SquareMatrix(n,K)        -- export matrix
        ++ \bs{}axiom\{matrix(qf)\} creates a square matrix
        ++ from the quadratic form \bs{}axiom\{qf\}.
      elt: (\$, DirectProduct(n,K)) -> K     -- export elt
        ++ \bs{}axiom\{qf(v)\} evaluates the quadratic form
        ++ \bs{}axiom\{qf\} on the vector \bs{}axiom\{v\},
        ++ producing a scalar.

    Implementation == SquareMatrix(n,K) add --The exports
      Rep := SquareMatrix(n,K)               --representation
      quadraticForm m ==                     --definition
        not symmetric? m => error
          "quadraticForm requires a symmetric matrix"
        m :: $
      matrix q == q :: Rep                   --definition
      elt(q,v) == dot(v, (matrix q * v))     --definition
```

Figure 13.1: The `QuadraticForm` domain.

members. The word "unconditionally" means that membership in a category does not depend on the values of the parameters to the domain constructor. This part thus defines the link between the domains and the category hierarchies given on the inside covers of this book. As described in 12.8 on page 917, it is this link that makes it possible for you to pass objects of the domains as arguments to other operations in Axiom.

Every `QuadraticForm` domain is declared to be unconditionally a member of category `AbelianGroup`. An abelian group is a collection of elements closed under addition. Every object x of an abelian group has an additive inverse y such that $x + y = 0$. The exports of an abelian group include 0, +, -, and scalar

multiplication by an integer. After asserting that `QuadraticForm` domains are abelian groups, it is possible to pass quadratic forms to algorithms that only assume arguments to have these abelian group properties.

In 12.11 on page 920, you saw that `Fraction(R)`, a member of `QuotientFieldCategory(R)`, is a member of `OrderedSet` if R is a member of `OrderedSet`. Likewise, from the `Exports` part of the definition of `ModMonic(R, S)`,

```
UnivariatePolynomialCategory(R) with
  if R has Finite then Finite
     ...
```

you see that `ModMonic(R, S)` is a member of `Finite` is R is.

The `Exports` part of a domain definition is the same kind of expression that can appear to the right of an `==` in a category definition. If a domain constructor is unconditionally a member of two or more categories, a *Join* form is used. The `Exports` part of the definition of `FlexibleArray(S)` reads, for example:

```
Join(ExtensibleLinearAggregate(S),
     OneDimensionalArrayAggregate(S)) with...
```

13.4   A Demo

Before looking at the *Implementation* part of `QuadraticForm`, let's try some examples.

Build a domain QF.

```
QF := QuadraticForm(2,Fraction Integer)
```

Define a matrix to be used to construct a quadratic form.

```
A := matrix [ [-1,1/2],[1/2,1] ]
```

Construct the quadratic form. A package call `$QF` is necessary since there are other `QuadraticForm` domains.

```
q :  QF := quadraticForm(A)
```

Looks like a matrix. Try computing the number of rows. Axiom won't let you.

```
nrows q
```

Create a direct product element v. A package call is again necessary, but Axiom understands your list as denoting a vector.

```
v := directProduct([2,-1])$DirectProduct(2,Fraction Integer)
```

Compute the product $v^T A v$.

```
q.v
```

What is 3 times q minus q plus q?

```
3*q-q+q
```

13.5 Browse

The Browse facility of HyperDoc is useful for investigating the properties of domains, packages, and categories. From the main HyperDoc menu, move your mouse to **Browse** and click on the left mouse button. This brings up the Browse first page. Now, with your mouse pointer somewhere in this window, enter the string "quadraticform" into the input area (all lower case letters will do). Move your mouse to **Constructors** and click. Up comes a page describing `QuadraticForm`.

From here, click on **Description**. This gives you a page that includes a part labeled by "*Description:*". You also see the types for arguments n and K displayed as well as the fact that `QuadraticForm` returns an `AbelianGroup`. You can go and experiment a bit by selecting `Field` with your mouse. Eventually, use the "UP" button several times to return to the first page on `QuadraticForm`.

Select **Operations** to get a list of operations for `QuadraticForm`. You can select an operation by clicking on it to get an individual page with information about that operation. Or you can select the buttons along the bottom to see alternative views or get additional information on the operations. Then return to the page on `QuadraticForm`.

Select **Cross Reference** to get another menu. This menu has buttons for **Parents**, **Ancestors**, and others. Clicking on **Parents**, you see that `QuadraticForm` has one parent `AbelianMonoid`.

13.6 Representation

The `Implementation` part of an Axiom capsule for a domain constructor uses the special variable *Rep* to identify the lower level data type used to represent

the objects of the domain. The *Rep* for quadratic forms is `SquareMatrix(n,`
`K)`. This means that all objects of the domain are required to be n by n matrices
with elements from **K**.

The code for `quadraticForm` in Figure 13.1 on page 925 checks that the matrix
is symmetric and then converts it to `$`, which means, as usual, "this domain."
Such explicit conversions are generally required by the compiler. Aside from
checking that the matrix is symmetric, the code for this function essentially
does nothing. The `m :: $` on line 28 coerces m to a quadratic form. In fact,
the quadratic form you created in step (3) of 13.4 on page 926 is just the matrix
you passed it in disguise! Without seeing this definition, you would not know
that. Nor can you take advantage of this fact now that you do know! When
we try in the next step of 13.4 on page 926 to regard q as a matrix by asking
for **nrows**, the number of its rows, Axiom gives you an error message saying, in
effect, "Good try, but this won't work!"

The definition for the **matrix** function could hardly be simpler: it just returns
its argument after explicitly coercing its argument to a matrix. Since the argu-
ment is already a matrix, this coercion does no computation.

Within the context of a capsule, an object of `$` is regarded both as a quadratic
form *and* as a matrix.[1] This makes the definition of $q.v$ easy—it just calls the
dot product from `DirectProduct` to perform the indicated operation.

13.7   Multiple Representations

To write functions that implement the operations of a domain, you want to
choose the most computationally efficient data structure to represent the ele-
ments of your domain.

A classic problem in computer algebra is the optimal choice for an internal rep-
resentation of polynomials. If you create a polynomial, say $3x^2 + 5$, how does
Axiom hold this value internally? There are many ways. Axiom has nearly a
dozen different representations of polynomials, one to suit almost any purpose.
Algorithms for solving polynomial equations work most efficiently with poly-
nomials represented one way, whereas those for factoring polynomials are most
efficient using another. One often-used representation is a list of terms, each
term consisting of exponent-coefficient records written in the order of decreas-
ing exponents. For example, the polynomial $3x^2 + 5$ is represented by the list
$[[e : 2, c : 3], [e : 0, c : 5]]$.

What is the optimal data structure for a matrix? It depends on the application.
For large sparse matrices, a linked-list structure of records holding only the non-
zero elements may be optimal. If the elements can be defined by a simple formula
$f(i, j)$, then a compiled function for f may be optimal. Some programmers

---

[1] In case each of `$` and *Rep* have the same named operation available, the one from $ takes
precedence. Thus, if you want the one from `Rep`, you must package call it using a `$Rep` suffix.

prefer to represent ordinary matrices as vectors of vectors. Others prefer to represent matrices by one big linear array where elements are accessed with linearly computable indexes.

While all these simultaneous structures tend to be confusing, Axiom provides a helpful organizational tool for such a purpose: categories. `PolynomialCategory`, for example, provides a uniform user interface across all polynomial types. Each kind of polynomial implements functions for all these operations, each in its own way. If you use only the top-level operations in `PolynomialCategory` you usually do not care what kind of polynomial implementation is used.

Within a given domain, however, you define (at most) one representation.[2] If you want to have multiple representations (that is, several domains, each with its own representation), use a category to describe the `Exports`, then define separate domains for each representation.

13.8   Add Domain

The capsule part of `Implementation` defines functions that implement the operations exported by the domain—usually only some of the operations. In our demo in 13.4 on page 926, we asked for the value of $3 * q - q + q$. Where do the operations `*`, `+`, and `-` come from? There is no definition for them in the capsule!

The `Implementation` part of a definition can optionally specify an "add-domain" to the left of an `add` (for `QuadraticForm`, defines `SquareMatrix(n,K)` is the add-domain). The meaning of an add-domain is simply this: if the capsule part of the `Implementation` does not supply a function for an operation, Axiom goes to the add-domain to find the function. So do $*$, $+$ and $-$ (from QuadraticForm) come from `SquareMatrix(n,K)`?

13.9   Defaults

In Chapter 11 on page 897, we saw that categories can provide default implementations for their operations. How and when are they used? When Axiom finds that `QuadraticForm(2, Fraction Integer)` does not implement the operations `*`, `+`, and `-`, it goes to `SquareMatrix(2,Fraction Integer)` to find it. As it turns out, `SquareMatrix(2, Fraction Integer)` does not implement *any* of these operations!

What does Axiom do then? Here is its overall strategy. First, Axiom looks for a function in the capsule for the domain. If it is not there, Axiom looks in

---

[2]You can make that representation a `Union` type, however. See 2.5 on page 149 for examples of unions.

the add-domain for the operation. If that fails, Axiom searches the add-domain of the add-domain, and so on. If all those fail, it then searches the default packages for the categories of which the domain is a member. In the case of `QuadraticForm`, it searches `AbelianGroup`, then its parents, grandparents, and so on. If this fails, it then searches the default packages of the add-domain. Whenever a function is found, the search stops immediately and the function is returned. When all fails, the system calls **error** to report this unfortunate news to you. To find out the actual order of constructors searched for `QuadraticForm`, consult Browse: from the `QuadraticForm`, click on `Cross Reference`, then on `Lineage`.

Let's apply this search strategy for our example $3 * q - q + q$. The scalar multiplication comes first. Axiom finds a default implementation in `AbelianGroup&`. Remember from 12.6 on page 915 that `SemiGroup` provides a default definition for x^n by repeated squaring? `AbelianGroup` similarly provides a definition for nx by repeated doubling.

But the search of the defaults for `QuadraticForm` fails to find any `+` or `*` in the default packages for the ancestors of `QuadraticForm`. So it now searches among those for `SquareMatrix`. Category `MatrixCategory`, which provides a uniform interface for all matrix domains, is a grandparent of `SquareMatrix` and has a capsule defining many functions for matrices, including matrix addition, subtraction, and scalar multiplication. The default package `MatrixCategory&` is where the functions for $+$ and $-$ (from QuadraticForm) come from.

You can use Browse to discover where the operations for `QuadraticForm` are implemented. First, get the page describing `QuadraticForm`. With your mouse somewhere in this window, type a "2", press the $\boxed{\textbf{Tab}}$ key, and then enter "Fraction Integer" to indicate that you want the domain `QuadraticForm(2, Fraction Integer)`. Now click on **Operations** to get a table of operations and on `*` to get a page describing the `*` operation. Finally, click on **implementation** at the bottom.

13.10   Origins

Aside from the notion of where an operation is implemented, a useful notion is the *origin* or "home" of an operation. When an operation (such as **quadraticForm**) is explicitly exported by a domain (such as `QuadraticForm`), you can say that the origin of that operation is that domain. If an operation is not explicitly exported from a domain, it is inherited from, and has as origin, the (closest) category that explicitly exports it. The operations $+$ and $-$ (from AbelianMonoid) of `QuadraticForm`, for example, are inherited from `AbelianMonoid`. As it turns out, `AbelianMonoid` is the origin of virtually every `+` operation in Axiom!

Again, you can use Browse to discover the origins of operations. From the Browse page on `QuadraticForm`, click on **Operations**, then on **origins** at the bottom of the page.

The origin of the operation is the *only* place where on-line documentation is given. However, you can re-export an operation to give it special documentation. Suppose you have just invented the world's fastest algorithm for inverting matrices using a particular internal representation for matrices. If your matrix domain just declares that it exports `MatrixCategory`, it exports the **inverse** operation, but the documentation the user gets from Browse is the standard one from `MatrixCategory`. To give your version of **inverse** the attention it deserves, simply export the operation explicitly with new documentation. This redundancy gives **inverse** a new origin and tells Browse to present your new documentation.

13.11 Short Forms

In Axiom, a domain could be defined using only an add-domain and no capsule. Although we talk about rational numbers as quotients of integers, there is no type `RationalNumber` in Axiom. To create such a type, you could compile the following "short-form" definition:

```
RationalNumber() == Fraction(Integer)
```

The `Exports` part of this definition is missing and is taken to be equivalent to that of `Fraction(Integer)`. Because of the add-domain philosophy, you get precisely what you want. The effect is to create a little stub of a domain. When a user asks to add two rational numbers, Axiom would ask `RationalNumber` for a function implementing this `+`. Since the domain has no capsule, the domain then immediately sends its request to `Fraction (Integer)`.

The short form definition for domains is used to define such domains as `MultivariatePolynomial`:

```
MultivariatePolynomial(vl: List Symbol, R: Ring) ==
   SparseMultivariatePolynomial(R,
      OrderedVariableList vl)
```

13.12 Example 1: Clifford Algebra

Now that we have `QuadraticForm` available, let's put it to use. Given some quadratic form Q described by an n by n matrix over a field K, the domain `CliffordAlgebra(n, K, Q)` defines a vector space of dimension 2^n over K. This is an interesting domain since complex numbers, quaternions, exterior algebras and spin algebras are all examples of Clifford algebras.

The basic idea is this: the quadratic form Q defines a basis $e_1, e_2 \ldots, e_n$ for the vector space K^n—the direct product of K with itself n times. From this,

the Clifford algebra generates a basis of 2^n elements given by all the possible products of the e_i in order without duplicates, that is,

1, e_1, e_2, e_1e_2, e_3, e_1e_3, e_2e_3, e_1e_2, e_3, and so on.

The algebra is defined by the relations

$$
\begin{aligned}
e_i\, e_i &= Q(e_i) \\
e_i\, e_j &= -e_j\, e_i \quad \text{for } i \neq j
\end{aligned}
$$

Now look at the snapshot of its definition given in Figure 13.2 on page 933. Lines 9-10 show part of the definitions of the `Exports`. A Clifford algebra over a field K is asserted to be a ring, an algebra over K, and a vector space over K. Its explicit exports include $e(n)$, which returns the n-th unit element.

The `Implementation` part begins by defining a local variable *Qeelist* to hold the list of all $q.v$ where v runs over the unit vectors from 1 to the dimension n. Another local variable *dim* is set to 2^n, computed once and for all. The representation for the domain is `PrimitiveArray(K)`, which is a basic array of elements from domain K. Line 18 defines *New* as shorthand for the more lengthy expression $new(dim, 0\$K)\$Rep$, which computes a primitive array of length 2^n filled with 0's from domain K.

Lines 19-22 define the sum of two elements x and y straightforwardly. First, a new array of all 0's is created, then filled with the sum of the corresponding elements. Indexing for primitive arrays starts at 0. The definition of the product of x and y first requires the definition of a local function **addMonomProd**. Axiom knows it is local since it is not an exported function. The types of all local functions must be declared.

13.13   Example 2: Building A Query Facility

We now turn to an entirely different kind of application, building a query language for a database.

Here is the practical problem to solve. The Browse facility of Axiom has a database for all operations and constructors which is stored on disk and accessed by HyperDoc. For our purposes here, we regard each line of this file as having eight fields: `class`, `name`, `type`, `nargs`, `exposed`, `kind`, `origin`, and `condition`. Here is an example entry:

```
o'determinant'$->R'1'x'd'Matrix(R)'has(R,commutative("*"))
```

In English, the entry means:

> The operation **determinant**: $ → R with *1* argument, is
> *exposed* and is exported by *domain* `Matrix(R)` if `R has`
> `commutative("*")`.

```
NNI ==> NonNegativeInteger
PI  ==> PositiveInteger

CliffordAlgebra(n,K,q): Exports == Implementation where
    n: PI
    K: Field
    q: QuadraticForm(n, K)

    Exports == Join(Ring,Algebra(K),VectorSpace(K)) with
      e: PI -> $
          ...

    Implementation == add
      Qeelist :=
        [q.unitVector(i::PI) for i in 1..n]
      dim     :=  2**n
      Rep     := PrimitiveArray K
      New ==> new(dim, 0$K)$Rep
      x + y ==
        z := New
        for i in 0..dim-1 repeat z.i := x.i + y.i
        z
      addMonomProd: (K, NNI, K, NNI, $) -> $
      addMonomProd(c1, b1, c2, b2, z) ==  ...
      x * y ==
        z := New
        for ix in 0..dim-1 repeat
          if x.ix \notequal{} 0 then for iy in 0..dim-1 repeat
            if y.iy \notequal{} 0
            then addMonomProd(x.ix,ix,y.iy,iy,z)
          z
          ...
```

Figure 13.2: Part of the `CliffordAlgebra` domain.

Our task is to create a little query language that allows us to get useful information from this database.

13.13.1   A Little Query Language

First we design a simple language for accessing information from the database. We have the following simple model in mind for its design. Think of the database as a box of index cards. There is only one search operation—it takes the name of a field and a predicate (a boolean-valued function) defined on the fields of

the index cards. When applied, the search operation goes through the entire box selecting only those index cards for which the predicate is `true`. The result of a search is a new box of index cards. This process can be repeated again and again.

The predicates all have a particularly simple form: *symbol* `=` *pattern*, where *symbol* designates one of the fields, and *pattern* is a "search string"—a string that may contain a "`*`" as a wildcard. Wildcards match any substring, including the empty string. Thus the pattern "`"*ma*t`" matches "`"mat`","`doormat`" and "`smart`".

To illustrate how queries are given, we give you a sneak preview of the facility we are about to create.

Extract the database of all Axiom operations.

```
ops := getDatabase("o")
```

How many exposed three-argument **map** operations involving streams?

```
ops.(name="map").(nargs="3").(type="*Stream*")
```

As usual, the arguments of **elt** (`.`) associate to the left. The first **elt** produces the set of all operations with name `map`. The second **elt** produces the set of all map operations with three arguments. The third **elt** produces the set of all three-argument map operations having a type mentioning `Stream`.

Another thing we'd like to do is to extract one field from each of the index cards in the box and look at the result. Here is an example of that kind of request.

What constructors explicitly export a **determinant** operation?

```
elt(elt(elt(elt(ops,name="determinant"),origin),sort),unique)
```

The first **elt** produces the set of all index cards with name `determinant`. The second **elt** extracts the `origin` component from each index card. Each origin component is the name of a constructor which directly exports the operation represented by the index card. Extracting a component from each index card produces what we call a *datalist*. The third **elt**, `sort`, causes the datalist of origins to be sorted in alphabetic order. The fourth, `unique`, causes duplicates to be removed.

Before giving you a more extensive demo of this facility, we now build the necessary domains and packages to implement it.

13.13.2   The Database Constructor

We work from the top down. First, we define a database, our box of index cards, as an abstract datatype. For sake of illustration and generality, we assume that

an index card is some type S, and that a database is a box of objects of type S. Here is the Axiom program defining the `Database` domain.

```
PI ==> PositiveInteger
Database(S): Exports == Implementation where
  S: Object with
    elt: (\$, Symbol) -> String
    display: \$ -> Void
    fullDisplay: \$ -> Void

  Exports == with
    elt: (\$,QueryEquation) -> \$            Select by an equation
    elt: (\$, Symbol) -> DataList String     Select by a field name
    "+": (\$,\$) -> \$                       Combine two databases
    "-": (\$,\$) -> \$                       Subtract one from another
    display: \$ -> Void                      A brief database display
    fullDisplay: \$ -> Void                  A full database display
    fullDisplay: (\$,PI,PI) -> Void          A selective display
    coerce: \$ -> OutputForm                 Display a database
  Implementation == add
      ...
```

The domain constructor takes a parameter S, which stands for the class of index cards. We describe an index card later. Here think of an index card as a string which has the eight fields mentioned above.

First, we tell Axiom what operations we are going to require from index cards. We need an **elt** to extract the contents of a field (such as `name` and `type`) as a string. For example, *c.name* returns a string that is the content of the *name* field on the index card c. We need to display an index card in two ways: **display** shows only the name and type of an operation; **fullDisplay** displays all fields. The display operations return no useful information and thus have return type `Void`.

Next, we tell Axiom what operations the user can apply to the database. This part defines our little query language. The most important operation is `db .` `field = pattern` which returns a new database, consisting of all index cards of `db` such that the *field* part of the index card is matched by the string pattern called *pattern*. The expression `field = pattern` is an object of type `QueryEquation` (defined in the next section).

Another **elt** is needed to produce a `DataList` object. Operation + is to merge two databases together; - is used to subtract away common entries in a second database from an initial database. There are three display functions. The **fullDisplay** function has two versions: one that prints all the records, the other that prints only a fixed number of records. A **coerce** to `OutputForm` creates a display object.

The `Implementation` part of `Database` is straightforward.

```
Implementation == add
  s: Symbol
  Rep := List S
  elt(db,equation) == ...
  elt(db,key) == [x.key for x in db]::DataList(String)
  display(db) ==  for x in db repeat display x
  fullDisplay(db) == for x in db repeat fullDisplay x
  fullDisplay(db, n, m) == for x in db for i in 1..m
    repeat
      if i >= n then fullDisplay x
  x+y == removeDuplicates! merge(x,y)
  x-y == mergeDifference(copy(x::Rep),
                         y::Rep)$MergeThing(S)
  coerce(db): OutputForm == (#db):: OutputForm
```

The database is represented by a list of elements of S (index cards). We leave
the definition of the first **elt** operation (on line 4) until the next section. The
second **elt** collects all the strings with field name *key* into a list. The **display**
function and first **fullDisplay** function simply call the corresponding functions
from S. The second **fullDisplay** function provides an efficient way of printing
out a portion of a large list. The + is defined by using the existing **merge**
operation defined on lists, then removing duplicates from the result. The −
operation requires writing a corresponding subtraction operation. A package
`MergeThing` (not shown) provides this.

The **coerce** function converts the database to an `OutputForm` by computing
the number of index cards. This is a good example of the independence of the
representation of an Axiom object from how it presents itself to the user. We
usually do not want to look at a database—but do care how many "hits" we
get for a given query. So we define the output representation of a database to
be simply the number of index cards our query finds.

13.13.3   Query Equations

The predicate for our search is given by an object of type `QueryEquation`.
Axiom does not have such an object yet so we have to invent it.

```
QueryEquation(): Exports == Implementation where
  Exports == with
    equation: (Symbol, String) -> \$
    variable: \$ -> Symbol
    value:\ \ \ \ \$ -> String

  Implementation == add
    Rep := Record(var:Symbol, val:String)
    equation(x, s) == [x, s]
```

```
variable q == q.var
value\ \ \ \ q == q.val
```

Axiom converts an input expression of the form $a = b$ to *equation*(a, b). Our equations always have a symbol on the left and a string on the right. The `Exports` part thus specifies an operation **equation** to create a query equation, and **variable** and **value** to select the left- and right-hand sides. The `Implementation` part uses `Record` for a space-efficient representation of an equation.

Here is the missing definition for the **elt** function of `Database` in the last section:

```
elt(db,eq) ==
  field\  := variable eq
  value := value eq
  [x for x in db | matches?(value,x.field)]
```

Recall that a database is represented by a list. Line 4 simply runs over that list collecting all elements such that the pattern (that is, *value*) matches the selected field of the element.

13.13.4 DataLists

Type `DataList` is a new type invented to hold the result of selecting one field from each of the index cards in the box. It is useful to make datalists extensions of lists—lists that have special **elt** operations defined on them for sorting and removing duplicates.

```
DataList(S:OrderedSet) : Exports == Implementation where
  Exports == ListAggregate(S) with
    elt: ($,"unique") -> $
    elt: ($,"sort") -> $
    elt: ($,"count") -> NonNegativeInteger
    coerce: List S -> $

  Implementation ==  List(S) add
    Rep := List S
    elt(x,"unique") == removeDuplicates(x)
    elt(x,"sort") == sort(x)
    elt(x,"count") == #x
    coerce(x:List S) == x :: $
```

The `Exports` part asserts that datalists belong to the category `ListAggregate`. Therefore, you can use all the usual list operations on datalists, such as **first**, **rest**, and **concat**. In addition, datalists have four explicit operations. Besides

the three **elt** operations, there is a **coerce** operation that creates datalists from lists.

The `Implementation` part needs only to define four functions. All the rest are obtained from `List(S)`.

13.13.5   Index Cards

An index card comes from a file as one long string. We define functions that extract substrings from the long string. Each field has a name that is passed as a second argument to **elt**.

```
IndexCard() == Implementation where
  Exports == with
    elt: ($, Symbol) -> String
    display: $ -> Void
    fullDisplay: $ -> Void
    coerce: String -> $
  Implementation == String add ...
```

We leave the `Implementation` part to the reader. All operations involve straightforward string manipulations.

13.13.6   Creating a Database

We must not forget one important operation: one that builds the database in the first place! We'll name it **getDatabase** and put it in a package. This function is implemented by calling the Common Lisp function *getBrowseDatabase(s)* to get appropriate information from Browse. This operation takes a string indicating which lines you want from the database: "o" gives you all operation lines, and "k", all constructor lines. Similarly, "c", "d", and "p" give you all category, domain and package lines respectively.

```
OperationsQuery(): Exports == Implementation where
  Exports == with
    getDatabase: String -> Database(IndexCard)

  Implementation == add
    getDatabase(s) == getBrowseDatabase(s)$Lisp
```

We do not bother creating a special name for databases of index cards. `Database (IndexCard)` will do. Notice that we used the package `OperationsQuery` to create, in effect, a new kind of domain: `Database(IndexCard)`.

13.13.7 Putting It All Together

To create the database facility, you put all these constructors into one file.[3] At the top of the file put `)abbrev` commands, giving the constructor abbreviations you created.

```
)abbrev domain  ICARD   IndexCard
)abbrev domain  QEQUAT  QueryEquation
)abbrev domain  MTHING  MergeThing
)abbrev domain  DLIST   DataList
)abbrev domain  DBASE   Database
)abbrev package OPQUERY OperationsQuery
```

With all this in **alql.spad**, for example, compile it using

```
)compile alql
```

and then load each of the constructors:

```
)load ICARD QEQUAT MTHING DLIST DBASE OPQUERY
```

You are ready to try some sample queries.

13.13.8 Example Queries

Our first set of queries give some statistics on constructors in the current Axiom system.

How many constructors does Axiom have?

```
ks := getDatabase "k"
```

Break this down into the number of categories, domains, and packages.

```
[ks.(kind=k) for k in ["c","d","p"] ]
```

What are all the domain constructors that take no parameters?

```
elt(ks.(kind="d").(nargs="0"),name)
```

How many constructors have "Matrix" in their name?

---

[3]You could use separate files, but we are putting them all together because, organizationally, that is the logical thing to do.

```
mk := ks.(name="*Matrix*")
```

What are the names of those that are domains?

```
elt(mk.(kind="d"),name)
```

How many operations are there in the library?

```
o := getDatabase "o"
```

Break this down into categories, domains, and packages.

```
[o.(kind=k) for k in ["c","d","p"] ]
```

The query language is helpful in getting information about a particular oper-
ation you might like to apply. While this information can be obtained with
Browse, the use of the query database gives you data that you can manipulate
in the workspace.

How many operations have "eigen" in the name?

```
eigens := o.(name="*eigen*")
```

What are their names?

```
elt(eigens,name)
```

Where do they come from?

```
elt(elt(elt(eigens,origin),sort),unique)
```

The operations + and - are useful for constructing small databases and com-
bining them. However, remember that the only matching you can do is string
matching. Thus a pattern such as "*Matrix*" on the type field matches any
type containing `Matrix`, `MatrixCategory`, `SquareMatrix`, and so on.

How many operations mention "Matrix" in their type?

```
tm := o.(type="*Matrix*")
```

How many operations come from constructors with "Matrix" in their name?

```
fm := o.(origin="*Matrix*")
```

How many operations are in fm but not in tm?

```
fm-tm
```

Display the operations that both mention "Matrix" in their type and come from a constructor having "Matrix" in their name.

```
fullDisplay(fm-%)
```

How many operations involve matrices?

```
m := tm+fm
```

Display 4 of them.

```
fullDisplay(m, 202, 205)
```

How many distinct names of operations involving matrices are there?

```
elt(elt(elt(m,name),unique),count)
```

Chapter 14

Browse

This chapter discusses the Browse component of HyperDoc. We suggest you invoke Axiom and work through this chapter, section by section, following our examples to gain some familiarity with Browse.

14.1   The Front Page: Searching the Library

To enter Browse, click on **Browse** on the top level page of HyperDoc to get the *front page* of Browse.
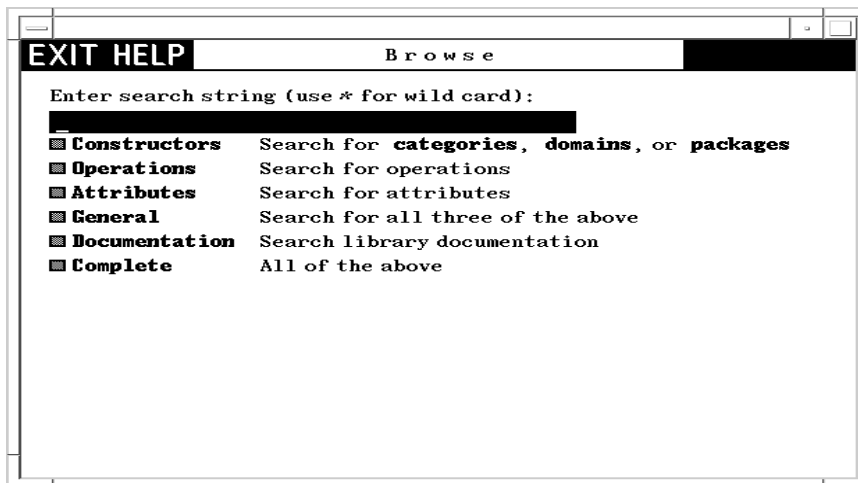
Figure 14.1: The Browse front page.

To use this page, you first enter a *search string* into the input area at the top,
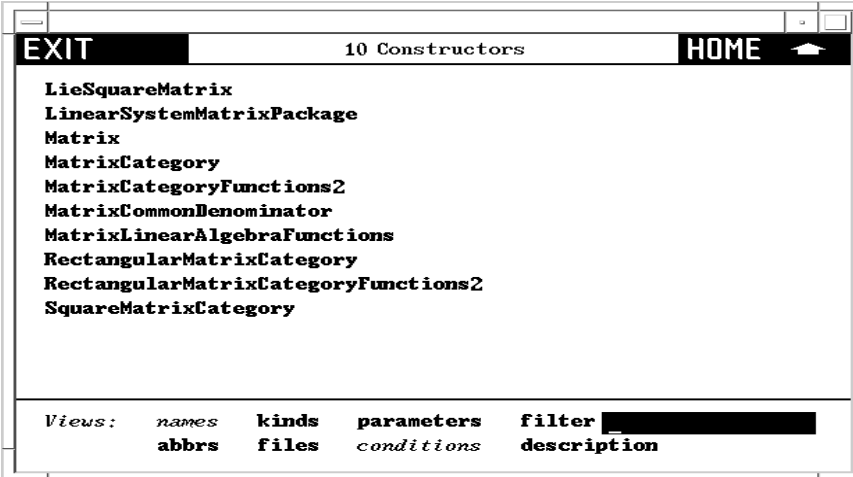
then click on one of the buttons below. We show the use of each of the buttons by example.

Constructors

First enter the search string `Matrix` into the input area and click on **Constructors**. What you get is the *constructor page* for `Matrix`. We show and describe this page in detail in 14.2 on page 947. By convention, Axiom does a case-insensitive search for a match. Thus `matrix` is just as good as `Matrix`, has the same effect as `MaTrix`, and so on. We recommend that you generally use small letters for names however. A search string with only capital letters has a special meaning (see 14.3.3 on page 966).

Click on [arrow button] to return to the Browse front page.

Use the symbol "`*`" in search strings as a *wild card.* A wild card matches any substring, including the empty string. For example, enter the search string `*matrix*` into the input area and click on **Constructors**.[1] What you get is a table of all constructors whose names contain the string "`matrix`."


```
EXIT                          10 Constructors              HOME  ←

    LieSquareMatrix
    LinearSystemMatrixPackage
    Matrix
    MatrixCategory
    MatrixCategoryFunctions2
    MatrixCommonDenominator
    MatrixLinearAlgebraFunctions
    RectangularMatrixCategory
    RectangularMatrixCategoryFunctions2
    SquareMatrixCategory



    Views:     names     kinds     parameters     filter ▮▮▮▮▮▮
               abbrs     files     conditions     description
```

Figure 14.2: Table of exposed constructors matching `*matrix*` .

All constructors containing the string are listed, whether exposed or unexposed. You can hide the names of the unexposed constructors by clicking on the **=**unexposed** button in the *Views* panel at the bottom of the window. (The button will change to **exposed** *only.*)

One of the names in this table is `Matrix`. Click on `Matrix`. What you get is again the constructor page for `Matrix`. As you see, Browse gives you a large

---

[1]To get only categories, domains, or packages, rather than all constructors, you can click on the corresponding button to the right of **Constructors**.

network of information in which there are many ways to reach the same pages.

Again click on the ▭ to return to the table of constructors whose names contain `matrix`. Below the table is a *Views* panel. This panel contains buttons that let you view constructors in different ways. To learn about views of constructors, skip to 14.2.3 on page 957.

Click on ▭ to return to the Browse front page.

Operations

Enter `*matrix` into the input area and click on **Operations**. This time you get a table of *operations* whose names end with `matrix` or `Matrix`.

```
 ┌──────────────────────────────────────────────────────────┐
 │ EXIT          13 Names for 22 Operations      HOME  ◄─    │
 │                                                          │
 │   createGenericMatrix          leftTraceMatrix           │
 │   diagonalMatrix               matrix                    │
 │   eigenMatrix                  rightTraceMatrix          │
 │   getMultiplicationMatrix      scalarMatrix              │
 │   integralDerivationMatrix     subMatrix                 │
 │   integralMatrix               traceMatrix               │
 │   inverseIntegralMatrix                                  │
 │                                                          │
 │                                                          │
 │                                                          │
 │                                                          │
 │   Views: names   signatures   parameters   filter ████   │
 │                  origins      conditions   description   │
 └──────────────────────────────────────────────────────────┘
```
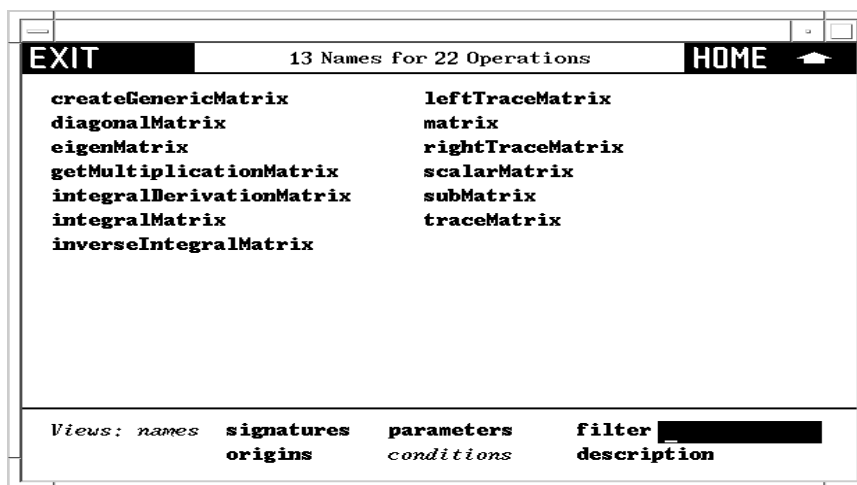
Figure 14.3: Table of operations matching `*matrix` .

If you select an operation name, you go to a page describing all the operations in Axiom of that name. At the bottom of an operation page is another kind of *Views* panel, one for operation pages. To learn more about these views, skip to 14.3.2 on page 961.

Click on ▭ to return to the Browse front page.

Attributes

This button gives you a table of attribute names that match the search string. Enter the search string `*` and click on **Attributes** to get a list of all system attributes.

Click on ▭ to return to the Browse front page.

Figure 14.4: Table of Axiom attributes.

Again there is a *Views* panel at the bottom with buttons that let you view the attributes in different ways.

General

This button does a general search for all constructor, operation, and attribute names matching the search string. Enter the search string `*matrix*` into the input area. Click on **General** to find all constructs that have `matrix` as a part of their name.

The summary gives you all the names under a heading when the number of entries is less than 10.

Click on [⬆] to return to the Browse front page.

Documentation

Again enter the search key `*matrix*` and this time click on **Documentation**. This search matches any constructor, operation, or attribute name whose documentation contains a substring matching `matrix`.

Click on [⬆] to return to the Browse front page.

Complete

This search combines both **General** and **Documentation**.

```
┌──────────────────────────────────────────────────────────┐
│ ─                                                    ▫ ☐  │
│ ┌──────────────────────────────────────────────────────┐ │
│ │ EXIT          35 entries match *matrix*     HOME  ◆  │ │
│ │  ■ 25 operations                                      │ │
│ │  ■ 3 categories                                       │ │
│ │     MatrixCategory              SquareMatrixCategory  │ │
│ │     RectangularMatrixCategory                         │ │
│ │  ■ 2 domains                                          │ │
│ │     LieSquareMatrix             Matrix                │ │
│ │  ■ 5 packages                                         │ │
│ │     LinearSystemMatrixPackage                         │ │
│ │     MatrixCategoryFunctions2                          │ │
│ │     MatrixCommonDenominator                           │ │
│ │     MatrixLinearAlgebraFunctions                      │ │
│ │     RectangularMatrixCategoryFunctions2               │ │
│ │                                                        │ │
│ └──────────────────────────────────────────────────────┘ │
└──────────────────────────────────────────────────────────┘
```
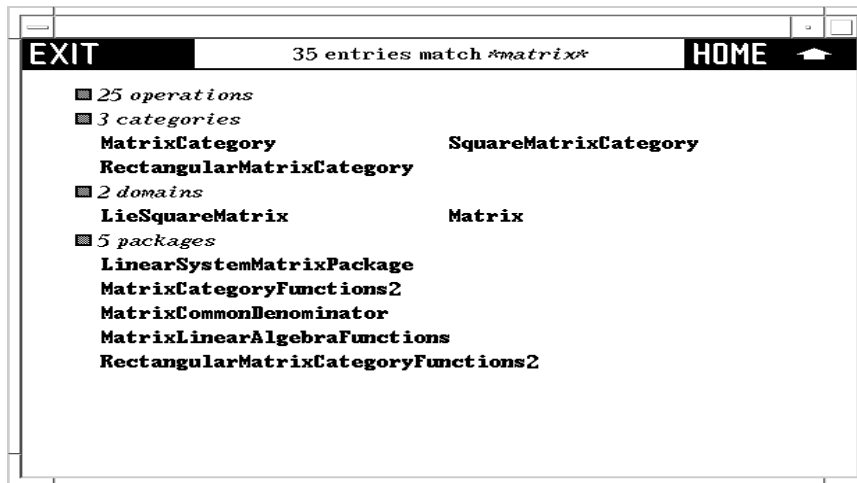
Figure 14.5: Table of all constructs matching *matrix* .

```
┌──────────────────────────────────────────────────────────┐
│ ─                                                    ▫ ☐  │
│ ┌──────────────────────────────────────────────────────┐ │
│ │ EXIT         141 entries match *matrix*     HOME  ◆  │ │
│ │  Documentation Summary: 141 mentions of *matrix*:     │ │
│ │  ■ 130 operations                                     │ │
│ │  ■ 3 categories                                       │ │
│ │     MatrixCategory              SquareMatrixCategory  │ │
│ │     RectangularMatrixCategory                         │ │
│ │  ■ 2 domains                                          │ │
│ │     LieSquareMatrix             Matrix                │ │
│ │  ■ 6 packages                                         │ │
│ │     CharacteristicPolynomialPackage                   │ │
│ │     LinearSystemMatrixPackage                         │ │
│ │     MatrixCategoryFunctions2                          │ │
│ │     MatrixCommonDenominator                           │ │
│ │     MatrixLinearAlgebraFunctions                      │ │
│ │     RectangularMatrixCategoryFunctions2               │ │
│ │                                                        │ │
│ └──────────────────────────────────────────────────────┘ │
└──────────────────────────────────────────────────────────┘
```

Figure 14.6: Table of constructs with documentation matching *matrix* .

14.2   The Constructor Page

In this section we look in detail at a constructor page for domain `Matrix`. Enter `matrix` into the input area on the main Browse page and click on **Constructors**.

The header part tells you that `Matrix` has abbreviation `MATRIX` and one argument called `R` that must be a domain of category `Ring`. Just what domains can be arguments of `Matrix`? To find this out, click on the `R` on the second line of the heading. What you get is a table of all acceptable domain parameter values

```
┌─────────────────────────────────────────────────────────────┐
│  ──                                                    ▫ □   │
│ ┌───────────────────────────────────────────────────────┐   │
│ │ EXIT          176 entries match *matrix*      HOME  ◄► │   │
│ │  Construct Summary: 35 names match:                ▲  │   │
│ │    ■ 25 operations                                    │   │
│ │    ■ 3 categories                                     │   │
│ │      MatrixCategory              SquareMatrixCategory │   │
│ │      RectangularMatrixCategory                        │   │
│ │    ■ 2 domains                                        │   │
│ │      LieSquareMatrix             Matrix               │   │
│ │    ■ 5 packages                                       │   │
│ │      LinearSystemMatrixPackage                        │   │
│ │      MatrixCategoryFunctions2                         │   │
│ │      MatrixCommonDenominator                          │   │
│ │      MatrixLinearAlgebraFunctions                     │   │
│ │      RectangularMatrixCategoryFunctions2              │   │
│ │  Documentation Summary: 141 mentions of matrix:       │   │
│ │    ■ 130 operations                                   │   │
│ │    ■ 3 categories                                  ▼  │   │
│ └───────────────────────────────────────────────────────┘   │
└─────────────────────────────────────────────────────────────┘
```

Figure 14.7: Table summarizing complete search for pattern `*matrix*` .

```
┌─────────────────────────────────────────────────────────────┐
│  ──                                                    ▫ □   │
│ ┌───────────────────────────────────────────────────────┐   │
│ │ EXIT              Domain Matrix(R)            HOME  ◄► │   │
│ │  Matrix has abbreviation MATRIX and takes one argument: │  │
│ │    R, a domain of category Ring                       │   │
│ │  The source code for the constructor is found in matrix.spad. │
│ │  ■ Description         General description            │   │
│ │  ■ Operations          All exported operations        │   │
│ │  ■ Attributes          All exported attributes        │   │
│ │  ■ Examples            Examples illustrating use      │   │
│ │  ■ Exports             Explicit categories and operations │ │
│ │  ■ Cross Reference     Hierarchy and usage information │   │
│ │  ■ Related Operations  Those exported from related packages │ │
│ │                                                       │   │
│ │  For domain information, enter R = ▮_____   │   │
│ │                                                       │   │
│ │                                                       │   │
│ │                                                       │   │
│ └───────────────────────────────────────────────────────┘   │
└─────────────────────────────────────────────────────────────┘
```

Figure 14.8: Constructor page for `Matrix`.

of `R`, or a table of *rings* in Axiom.

Click on ◄► to return to the constructor page for `Matrix`.
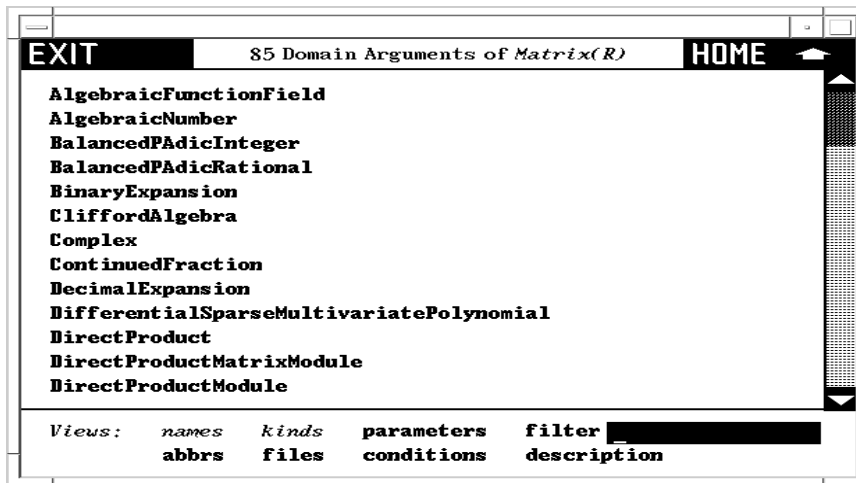
Figure 14.9: Table of acceptable domain parameters to `Matrix`.

If you have access to the source code of Axiom, the third line of the heading gives you the name of the source file containing the definition of `Matrix`. Click on it to pop up an editor window containing the source code of `Matrix`.
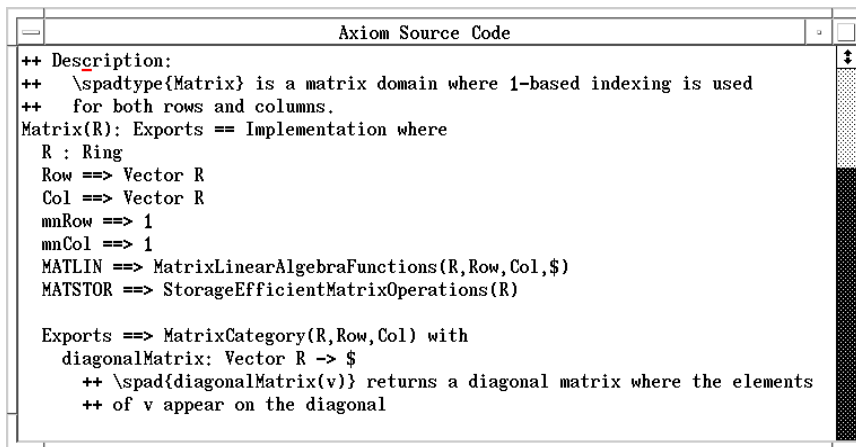
Figure 14.10: Source code for `Matrix`.

We recommend that you leave the editor window up while working through this chapter as you occasionally may want to refer to it.

14.2.1   Constructor Page Buttons

We examine each button on this page in order.

Description

Click here to bring up a page with a brief description of constructor `Matrix`. If you have access to system source code, note that these comments can be found directly over the constructor definition.
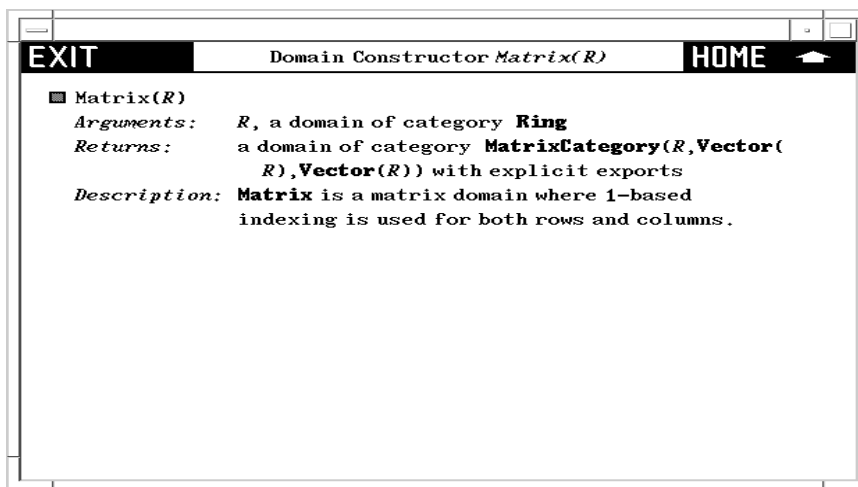
Figure 14.11: Description page for `Matrix`.

Operations

Click here to get a table of operations exported by `Matrix`. You may wish to widen the window to have multiple columns as below.

If you click on an operation name, you bring up a description page for the operations. For a detailed description of these pages, skip to .

Attributes

Click here to get a table of the two attributes exported by `Matrix`: **finiteAggregate** and **shallowlyMutable**. These are two computational properties that result from `Matrix` being regarded as a data structure.

Figure 14.12: Table of operations from `Matrix`.

Figure 14.13: Attributes from `Matrix`.

Examples

Click here to get an *examples page* with examples of operations to create and manipulate matrices.

Read through this section. Try selecting the various buttons. Notice that if you click on an operation name, such as **new**, you bring up a description page for that operation from `Matrix`.

Example pages have several examples of Axiom commands. Each example has
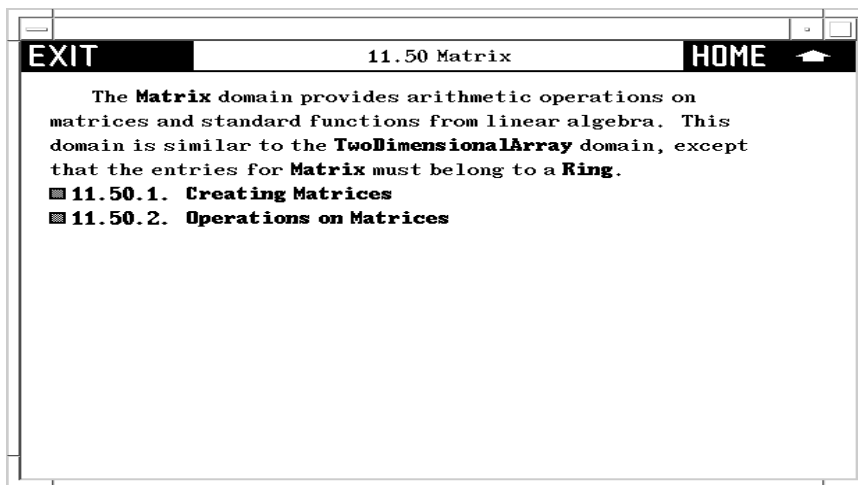
Figure 14.14: Example page for `Matrix`.

an active button to its left. Click on it! A pre-computed answer is pasted into the page immediately following the command. If you click on the button a second time, the answer disappears. This button thus acts as a toggle: "now you see it; now you don't."

Note also that the Axiom commands themselves are active. If you want to see Axiom execute the command, then click on it! A new Axiom window appears on your screen and the command is executed.

At the end of the page is generally a menu of buttons that lead you to further sections. Select one of these topics to explore its contents.

Exports

Click here to see a page describing the exports of `Matrix` exactly as described by the source code.

As you see, `Matrix` declares that it exports all the operations and attributes exported by category `MatrixCategory(R, Row, Col)`. In addition, two operations, **diagonalMatrix** and **inverse**, are explicitly exported.

To learn a little about the structure of Axiom, we suggest you do the following exercise.

Otherwise, go on to the next section.

`Matrix` explicitly exports only two operations. The other operations are thus exports of `MatrixCategory`. In general, operations are usually not explicitly exported by a domain. Typically they are inherited from several different categories. Let's find out from where the operations of `Matrix` come.
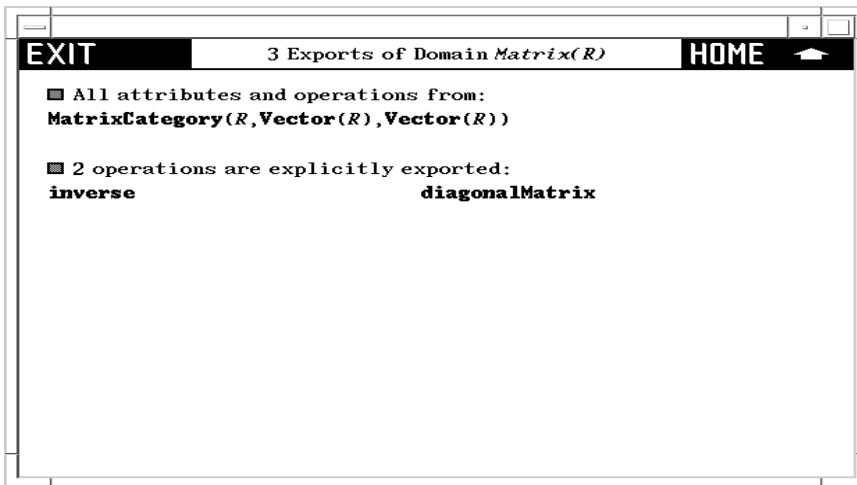
Figure 14.15: Exports of `Matrix`.

1. Click on `MatrixCategory`, then on **Exports**.   Here you see that `MatrixCategory` explicitly exports many matrix operations. Also, it inherits its operations from `TwoDimensionalArrayCategory`.

2. Click on `TwoDimensionalArrayCategory`, then on **Exports**. Here you see explicit operations dealing with rows and columns. In addition, it inherits operations from `HomogeneousAggregate`.

3. Click on [arrow] and then click on `Object`, then on **Exports**, where you see there are no exports.

4. Click on [arrow] repeatedly to return to the constructor page for `Matrix`.

Related Operations

Click here bringing up a table of operations that are exported by packages but not by `Matrix` itself.

To see a table of such packages, use the **Relatives** button on the **Cross Reference** page described next.

14.2.2   Cross Reference

Click on the **Cross Reference** button on the main constructor page for `Matrix`. This gives you a page having various cross reference information stored under the respective buttons.
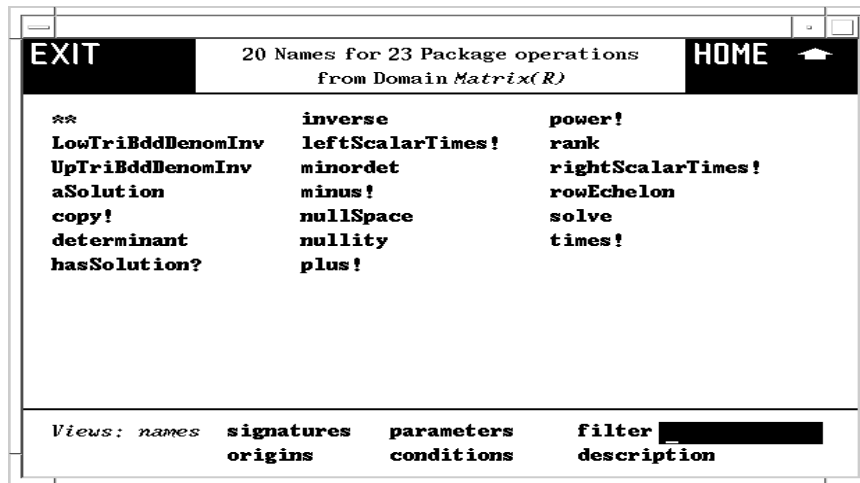
Figure 14.16: Related operations of `Matrix`.
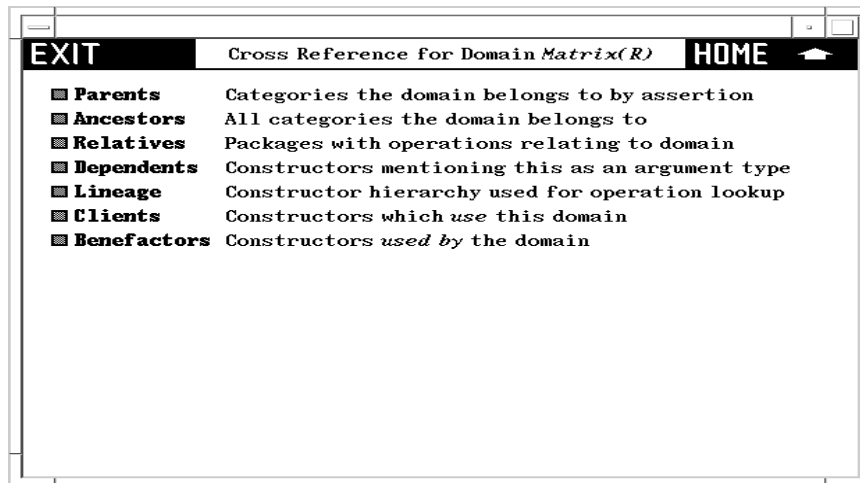
Figure 14.17: Cross-reference page for `Matrix`.

Parents

The parents of a domain are the same as the categories mentioned under the **Exports** button on the first page. Domain `Matrix` has only one parent but in general a domain can have any number.

Ancestors

The ancestors of a constructor consist of its parents, the parents of its parents, and so on. Did you perform the exercise in the last section under **Exports**? If so, you see here all the categories you found while ascending the **Exports** chain for `Matrix`.

Relatives

The relatives of a domain constructor are package constructors that provide operations in addition to those exported by the domain.

Try this exercise.

1. Click on **Relatives**, bringing up a list of packages.

2. Click on `LinearSystemMatrixPackage` bringing up its constructor page.[2]

3. Click on **Operations**. Here you see **rank**, an operation also exported by `Matrix` itself.

4. Click on **rank**. This **rank** has two arguments and thus is different from the **rank** from `Matrix`.

5. Click on ▲ to return to the list of operations for the package **LinearSystemMatrixPackage**.

6. Click on **solve** to bring up a **solve** for linear systems of equations.

7. Click on ▲ several times to return to the cross reference page for `Matrix`.

Dependents

The dependents of a constructor are those domains or packages that mention that constructor either as an argument or in its exports.

If you click on **Dependents** two entries may surprise you: `RectangularMatrix` and `SquareMatrix`. This happens because `Matrix`, as it turns out, appears in signatures of operations exported by these domains.

Lineage

The term *lineage* refers to the *search order* for functions. If you are an expert user or curious about how the Axiom system works, try the following exercise. Otherwise, you best skip this button and go on to **Clients**.

---

[2]You may want to widen your HyperDoc window to make what follows more legible.

Clicking on **Lineage** gives you a list of domain constructors: `InnerIndexedTwoDimensionalArray`, **MatrixCategory&**, **TwoDimensionalArrayCategory&**, **HomogeneousAggregate&**, **Aggregate&**. What are these constructors and how are they used?

We explain by an example. Suppose you create a matrix using the interpreter, then ask for its **rank**. Axiom must then find a function implementing the **rank** operation for matrices. The first place Axiom looks for **rank** is in the `Matrix` domain.

If not there, the lineage of `Matrix` tells Axiom where else to look. Associated with the matrix domain are five other lineage domains. Their order is important. Axiom first searches the first one, `InnerIndexedTwoDimensionalArray`. If not there, it searches the second **MatrixCategory&**. And so on.

Where do these *lineage constructors* come from? The source code for `Matrix` contains this syntax for the *function body* of `Matrix`:[3]

```
InnerIndexedTwoDimensionalArray(R,mnRow,mnCol,Row,Col)
   add ...
```

where the "`...`" denotes all the code that follows. In English, this means: "The functions for matrices are defined as those from `InnerIndexedTwoDimensionalArray` domain augmented by those defined in '`...`'," where the latter take precedence.

This explains `InnerIndexedTwoDimensionalArray`. The other names, those with names ending with an ampersand `&` are default packages for categories to which `Matrix` belongs. Default packages are ordered by the notion of "closest ancestor."

Clients

A client of `Matrix` is any constructor that uses `Matrix` in its implementation. For example, `Complex` is a client of `Matrix`; it exports several operations that take matrices as arguments or return matrices as values.[4]

Benefactors

A *benefactor* of `Matrix` is any constructor that `Matrix` uses in its implementation. This information, like that for clients, is gathered from run-time struc-

---

[3]`InnerIndexedTwoDimensionalArray` is a special domain implemented for matrix-like domains to provide efficient implementations of two-dimensional arrays. For example, domains of category `TwoDimensionalArrayCategory` can have any integer as their *minIndex*. Matrices and other members of this special "inner" array have their *minIndex* defined as 1.

[4]A constructor is a client of `Matrix` if it handles any matrix. For example, a constructor having internal (unexported) operations dealing with matrices is also a client.

tures.[5]

Cross reference pages for categories have some different buttons on them. Starting with the constructor page of `Matrix`, click on `Ring` producing its constructor page. Click on **Cross Reference**, producing the cross-reference page for `Ring`. Here are buttons **Parents** and **Ancestors** similar to the notion for domains, except for categories the relationship between parent and child is defined through *category extension.*

Children

Category hierarchies go both ways. There are children as well as parents. A child can have any number of parents, but always at least one. Every category is therefore a descendant of exactly one category: `Object.`

Descendants

These are children, children of children, and so on.

Category hierarchies are complicated by the fact that categories take parameters. Where a parameterized category fits into a hierarchy *may* depend on values of its parameters. In general, the set of categories in Axiom forms a *directed acyclic graph*, that is, a graph with directed arcs and no cycles.

Domains

This produces a table of all domain constructors that can possibly be rings (members of category `Ring`). Some domains are unconditional rings. Others are rings for some parameters and not for others. To find out which, select the **conditions** button in the views panel. For example, `DirectProduct(n, R)` is a ring if `R` is a ring.

14.2.3 Views Of Constructors

Below every constructor table page is a *Views* panel. As an example, click on **Cross Reference** from the constructor page of `Matrix`, then on **Benefactors** to produce a short table of constructor names.

The *Views* panel is at the bottom of the page. Two items, *names* and *conditions,* are in italics. Others are active buttons. The active buttons are those that give you useful alternative views on this table of constructors. Once you select a view, you notice that the button turns off (becomes italicized) so that you cannot reselect it.

---

[5]The benefactors exclude constructors such as `PrimitiveArray` whose operations macro-expand and so vanish from sight!

names

This view gives you a table of names. Selecting any of these names brings up the constructor page for that constructor.

abbrs

This view gives you a table of abbreviations, in the same order as the original constructor names. Abbreviations are in capitals and are limited to 7 characters. They can be used interchangeably with constructor names in input areas.

kinds

This view organizes constructor names into the three kinds: categories, domains and packages.

files

This view gives a table of file names for the source code of the constructors in alphabetic order after removing duplicates.

parameters

This view presents constructors with the arguments. This view of the benefactors of `Matrix` shows that `Matrix` uses as many as five different `List` domains in its implementation.

filter

This button is used to refine the list of names or abbreviations. Starting with the *names* view, enter `m*` into the input area and click on `filter`. You then get a shorter table with only the names beginning with `m`.

documentation

This gives you documentation for each of the constructors.

conditions

This page organizes the constructors according to predicates. The view is not available for your example page since all constructors are unconditional. For a table with conditions, return to the **Cross Reference** page for `Matrix`, click on **Ancestors**, then on **conditions** in the view panel. This page shows you

that `CoercibleTo(OutputForm)` and `SetCategory` are ancestors of `Matrix(R)` only if `R` belongs to category `SetCategory`.

14.2.4 Giving Parameters to Constructors

Notice the input area at the bottom of the constructor page. If you leave this blank, then the information you get is for the domain constructor `Matrix(R)`, that is, `Matrix` for an arbitrary underlying domain `R`.

In general, however, the exports and other information *do* usually depend on the actual value of `R`. For example, `Matrix` exports the **inverse** operation only if the domain `R` is a `Field`. To see this, try this from the main constructor page:

1. Enter `Integer` into the input area at the bottom of the page.

2. Click on **Operations**, producing a table of operations. Note the number of operation names that appear at the top of the page.

3. Click on to return to the constructor page.

4. Use the $\boxed{\textbf{Delete}}$ or $\boxed{\textbf{Backspace}}$ keys to erase `Integer` from the input area.

5. Click on **Operations** to produce a new table of operations. Look at the number of operations you get. This number is greater than what you had before. Find, for example, the operation **inverse**.

6. Click on **inverse** to produce a page describing the operation **inverse**. At the bottom of the description, you notice that the **Conditions** line says "R has `Field`." This operation is *not* exported by `Matrix(Integer)` since `Integer` is not a *field*.

   Try putting the name of a domain such as `Fraction Integer` (which is a field) into the input area, then clicking on **Operations**. As you see, the operation **inverse** is exported.

14.3 Miscellaneous Features of Browse

14.3.1 The Description Page for Operations

From the constructor page of `Matrix`, click on **Operations** to bring up the table of operations for `Matrix`.

Find the operation **inverse** in the table and click on it. This takes you to a page showing the documentation for this operation.

Here is the significance of the headings you see.
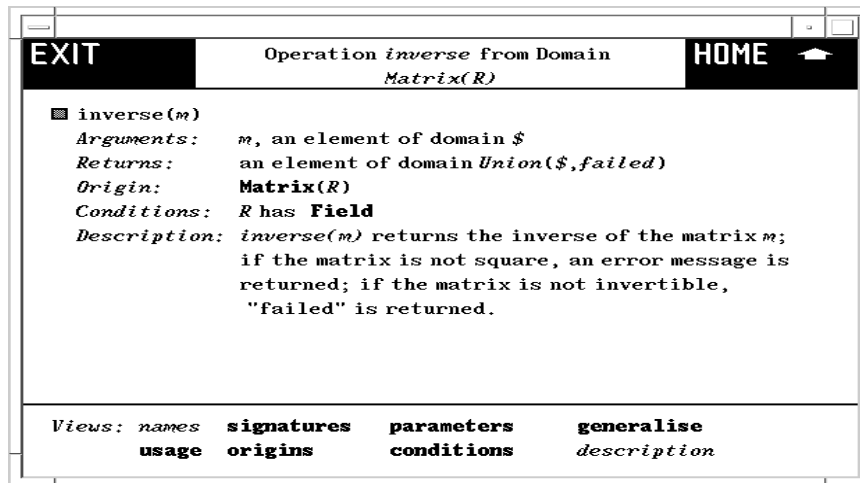
Figure 14.18: Operation **inverse** from `Matrix`.

Arguments

This lists each of the arguments of the operation in turn, paraphrasing the *signature* of the operation. As for signatures, a `$` is used to designate *this domain*, that is, `Matrix(R)`.

Returns

This describes the return value for the operation, analogous to the **Arguments** part.

Origin

This tells you which domain or category explicitly exports the operation. In this example, the domain itself is the *Origin*.

Conditions

This tells you that the operation is exported by `Matrix(R)` only if "`R` has `Field`," that is, "`R` is a member of category `Field`." When no **Conditions** part is given, the operation is exported for all values of `R`.

Description

Here are the `++` comments that appear in the source code of its *Origin*, here `Matrix`. You find these comments in the source code for `Matrix`.
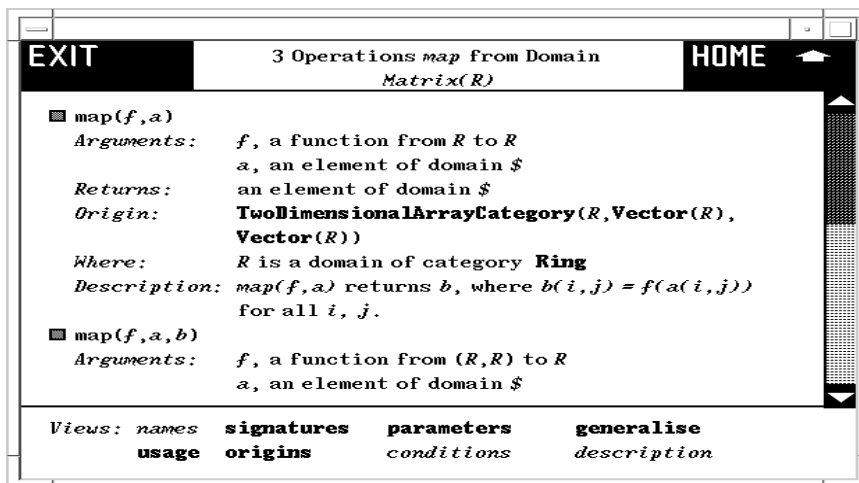
Figure 14.19: Operations **map** from `Matrix`.

Click on ![arrow] to return to the table of operations. Click on **map**. Here you find three different operations named **map**. This should not surprise you. Operations are identified by name and *signature*. There are three operations named **map**, each with different signatures. What you see is the *descriptions* view of the operations. If you like, select the button in the heading of one of these descriptions to get *only* that operation.

Where

This part qualifies domain parameters mentioned in the arguments to the operation.

14.3.2 Views of Operations

We suggest that you go to the constructor page for `Matrix` and click on **Operations** to bring up a table of operations with a *Views* panel at the bottom.

names

This view lists the names of the operations. Unlike constructors, however, there may be several operations with the same name. The heading for the page tells you the number of unique names and the number of distinct operations when these numbers are different.

filter

As for constructors, you can use this button to cut down the list of operations you are looking at. Enter, for example, `m*` into the input area to the right of **filter** then click on **filter**. As usual, any logical expression is permitted. For example, use

`*! or *?`

to get a list of destructive operations and predicates.

documentation

This gives you the most information: a detailed description of all the operations in the form you have seen before. Every other button summarizes these operations in some form.

signatures

This views the operations by showing their signatures.

parameters

This views the operations by their distinct syntactic forms with parameters.

origins

This organizes the operations according to the constructor that explicitly exports them.

conditions

This view organizes the operations into conditional and unconditional operations.

usage

This button is only available if your user-level is set to *development*. The **usage** button produces a table of constructors that reference this operation.[6]

---

[6]Axiom requires an especially long time to produce this table, so anticipate this when requesting this information.

implementation

This button is only available if your user-level is set to *development*. If you enter values for all domain parameters on the constructor page, then the **implementation** button appears in place of the **conditions** button. This button tells you what domains or packages actually implement the various operations.[7]

With your user-level set to *development*, we suggest you try this exercise. Return to the main constructor page for `Matrix`, then enter `Integer` into the input area at the bottom as the value of `R`. Then click on **Operations** to produce a table of operations. Note that the **conditions** part of the *Views* table is replaced by **implementation**. Click on **implementation**. After some delay, you get a page describing what implements each of the matrix operations, organized by the various domains and packages.
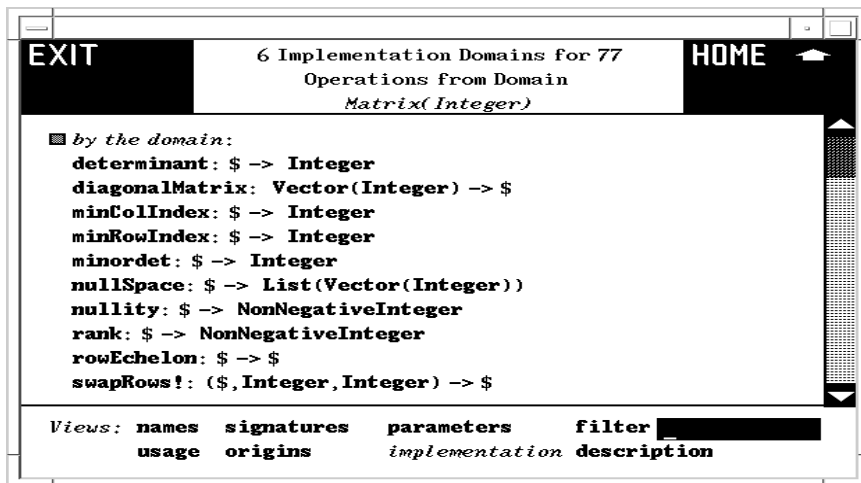
Figure 14.20: Implementation domains for `Matrix`.

generalize

This button only appears for an operation page of a constructor involving a unique operation name.

From an operations page for `Matrix`, select any operation name, say **rank**. In the views panel, the **filter** button is replaced by **generalize**. Click on it! What you get is a description of all Axiom operations named **rank**.[8]

---

[7]This button often takes a long time; expect a delay while you wait for an answer.

[8]If there were more than 10 operations of the name, you get instead a page with a *Views* panel at the bottom and the message to **Select a view below**. To get the descriptions of all these operations as mentioned above, select the **description** button.
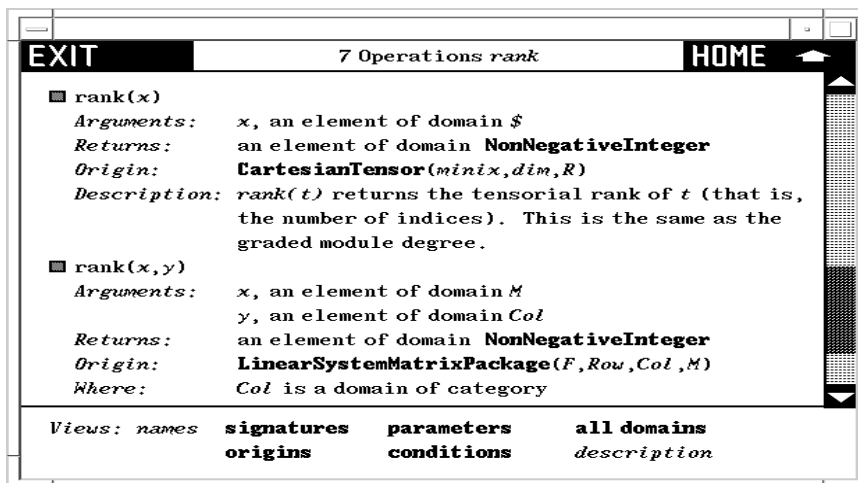
Figure 14.21: All operations named **rank** in Axiom.

all domains

This button only appears on an operation page resulting from a search from the front page of Browse or from selecting **generalize** from an operation page for a constructor.

Note that the **filter** button in the *Views* panel is replaced by **all domains**. Click on it to produce a table of *all* domains or packages that export a **rank** operation.
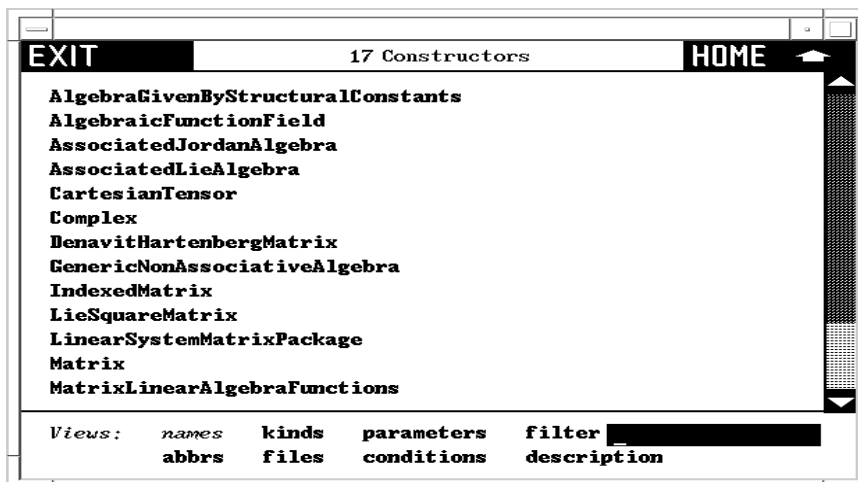
Figure 14.22: Table of all domains that export **rank**.

We note that this table specifically refers to all the **rank** operations shown in the preceding page. Return to the descriptions of all the **rank** operations and select one of them by clicking on the button in its heading. Select **all domains**. As you see, you have a smaller table of constructors. When there is only one constructor, you get the constructor page for that constructor.

14.3.3   Capitalization Convention

When entering search keys for constructors, you can use capital letters to search for abbreviations. For example, enter `UTS` into the input area and click on **Constructors**. Up comes a page describing `UnivariateTaylorSeries` whose abbreviation is `UTS`.

Constructor abbreviations always have three or more capital letters. For short constructor names (six letters or less), abbreviations are not generally helpful as their abbreviation is typically the constructor name in capitals. For example, the abbreviation for `Matrix` is `MATRIX`.

Abbreviations can also contain numbers. For example, `POLY2` is the abbreviation for constructor `PolynomialFunctions2`. For default packages, the abbreviation is the same as the abbreviation for the corresponding category with the "&" replaced by "-". For example, for the category default package **MatrixCategory&** the abbreviation is `MATCAT-` since the corresponding category `MatrixCategory` has abbreviation `MATCAT`.

Chapter 15

What's New in Axiom Version 2.0

Many things have changed in this new version of Axiom and we describe many of the more important topics here.

15.1   Important Things to Read First

If you have any private `.spad` files (that is, library files which were not shipped with Axiom) you will need to recompile them. For example, if you wrote the file `regress.spad` then you should issue `)compile regress.spad` before trying to use it.

The internal representation of `Union` has changed. This means that Axiom data saved with Release 1.x may not be readable by this Release. If you cannot recreate the saved data by recomputing in Release 2.0, please contact NAG for assistance.

15.2   The New Axiom Library Compiler

A new compiler is now available for Axiom. The programming language is referred to as the Aldor, and improves upon the old Axiom language in many ways. The `)compile` command has been upgraded to be able to invoke the new or old compilers. The language and the compiler are described in the hard-copy documentation which came with your Axiom system.

To ease the chore of upgrading your *.spad* files (old compiler) to *.as* files (new compiler), the `)compile` command has been given a `)translate` option. This invokes a special version of the old compiler which parses and analyzes your old

code and produces augmented code using the new syntax. Please be aware that the translation is not necessarily one hundred percent complete or correct. You should attempt to compile the output with the Aldor compiler and make any necessary corrections.

15.3 The NAG Library Link

The Nag Library link allows you to call NAG Fortran routines from within Axiom, passing Axiom objects as parameters and getting them back as results.

The Nag Library and, consequently, the link are divided into *chapters*, which cover different areas of numerical analysis. The statistical and sorting *chapters* of the Library, however, are not included in the link and various support and utility routines (mainly the F06 and X *chapters*) have been omitted.

Each *chapter* has a short (at most three-letter) name; for example, the *chapter* devoted to the solution of ordinary differential equations is called D02. When using the link via the HyperDoc interface. you will be presented with a complete menu of these *chapters*. The names of individual routines within each *chapter* are formed by adding three letters to the *chapter* name, so for example the routine for solving ODEs by Adams method is called **d02cjf**.

15.3.1 Interpreting NAG Documentation

Information about using the Nag Library in general, and about using individual routines in particular, can be accessed via HyperDoc. This documentation refers to the Fortran routines directly; the purpose of this subsection is to explain how this corresponds to the Axiom routines.

For general information about the Nag Library users should consult Essential Introduction to the NAG Foundation Library . The documentation is in ASCII format, and a description of the conventions used to represent mathematical symbols is given in Introduction to NAG On-Line Documentation . Advice about choosing a routine from a particular *chapter* can be found in the Chapter Documents .

Correspondence Between Fortran and Axiom types

The NAG documentation refers to the Fortran types of objects; in general, the correspondence to Axiom types is as follows.

- Fortran INTEGER corresponds to Axiom `Integer`.

- Fortran DOUBLE PRECISION corresponds to Axiom `DoubleFloat`.

- Fortran COMPLEX corresponds to Axiom `Complex DoubleFloat`.

- Fortran LOGICAL corresponds to Axiom `Boolean`.

- Fortran CHARACTER*(*) corresponds to Axiom `String`.

(Exceptionally, for NAG EXTERNAL parameters – ASPs in link parlance – REAL and COMPLEX correspond to `MachineFloat` and `MachineComplex`, respectively; see **??** on page **??**.)

The correspondence for aggregates is as follows.

- A one-dimensional Fortran array corresponds to an Axiom `Matrix` with one column.

- A two-dimensional Fortran ARRAY corresponds to an Axiom `Matrix`.

- A three-dimensional Fortran ARRAY corresponds to an Axiom `ThreeDimensionalMatrix`.

Higher-dimensional arrays are not currently needed for the Nag Library.

Arguments which are Fortran FUNCTIONs or SUBROUTINEs correspond to special ASP domains in Axiom. See **??** on page **??**.

Classification of NAG parameters

NAG parameters are classified as belonging to one (or more) of the following categories: `Input`, `Output`, `Workspace` or `External` procedure. Within `External` procedures a similar classification is used, and parameters may also be `Dummies`, or `User Workspace` (data structures not used by the NAG routine but provided for the convenience of the user).

When calling a NAG routine via the link the user only provides values for `Input` and `External` parameters.

The order of the parameters is, in general, different from the order specified in the Nag Library documentation. The Browser description for each routine helps in determining the correspondence. As a rule of thumb, `Input` parameters come first followed by `Input/Output` parameters. The `External` parameters are always found at the end.

IFAIL

NAG routines often return diagnostic information through a parameter called $ifail$. With a few exceptions, the principle is that on input $ifail$ takes one of the values $-1, 0, 1$. This determines how the routine behaves when it encounters an error:

- a value of 1 causes the NAG routine to return without printing an error message;

- a value of 0 causes the NAG routine to print an error message and abort;

- a value of -1 causes the NAG routine to return and print an error message.

The user is STRONGLY ADVISED to set $ifail$ to -1 when using the link. If $ifail$ has been set to 1 or -1 on input, then its value on output will determine the possible cause of any error. A value of 0 indicates successful completion, otherwise it provides an index into a table of diagnostics provided as part of the routine documentation (accessible via Browse).

15.3.2   Using the Link

The easiest way to use the link is via the HyperDoc interface . You will be presented with a set of fill-in forms where you can specify the parameters for each call. Initially, the forms contain example values, demonstrating the use of each routine (these, in fact, correspond to the standard NAG example program for the routine in question). For some parameters, these values can provide reasonable defaults; others, of course, represent data. When you change a parameter which controls the size of an array, the data in that array are reset to a "neutral" value – usually zero.

When you are satisfied with the values entered, clicking on the "Continue" button will display the Axiom command needed to run the chosen NAG routine with these values. Clicking on the "Do It" button will then cause Axiom to execute this command and return the result in the parent Axiom session, as described below. Note that, for some routines, multiple HyperDoc "pages" are required, due to the structure of the data. For these, returning to an earlier page causes HyperDoc to reset the later pages (this is a general feature of HyperDoc); in such a case, the simplest way to repeat a call, varying a parameter on an earlier page, is probably to modify the call displayed in the parent session.

An alternative approach is to call NAG routines directly in your normal Axiom session (that is, using the Axiom interpreter). Such calls return an object of type **Result**. As not all parameters in the underlying NAG routine are required in the AXIOM call (and the parameter ordering may be different), before calling a NAG routine you should consult the description of the Axiom operation in the Browser. (The quickest route to this is to type the routine name, in lower case, into the Browser's input area, then click on `Operations`.) The parameter names used coincide with NAG's, although they will appear here in lower case. Of course, it is also possible to become familiar with the Axiom form of a routine by first using it through the HyperDoc interface .

As an example of this mode of working, we can find a zero of a function, lying between 3 and 4, as follows:

```
answer:=c05adf(3.0,4.0,1.0e-5,0.0,-1,sin(X)::ASP1(F))
```

By default, **Result** only displays the type of returned values, since the amount
of information returned can be quite large. Individual components can be examined as follows:

```
answer . x
```

```
answer . ifail
```

In order to avoid conflict with names defined in the workspace, you can also
get the values by using the `String` type (the interpreter automatically coerces
them to `Symbol`)

```
answer "x"
```

It is possible to have Axiom display the values of scalar or array results automatically. For more details, see the commands **showScalarValues** and **showArrayValues**.

There is also a **.input** file for each NAG routine, containing Axiom interpreter
commands to set up and run the standard NAG example for that routine.

```
)read c05adf.input
```

15.3.3 Providing values for Argument Subprograms

There are a number of ways in which users can provide values for argument
subprograms (ASPs). At the top level the user will see that NAG routines
require an object from the `Union` of a `Filename` and an ASP.

For example **c05adf** requires an object of type
Union(fn: FileName,fp: Asp1 F)

```
)display operation c05adf
```

The user thus has a choice of providing the name of a file containing Fortran
source code, or of somehow generating the ASP within Axiom. If a filename is
specified, it is searched for in the *local* machine, i.e., the machine that Axiom is
running on.

Providing ASPs via `FortranExpression`

The `FortranExpression` domain is used to represent expressions which can
be translated into Fortran under certain circumstances. It is very simi-
lar to `Expression` except that only operators which exist in Fortran can
be used, and only certain variables can occur. For example the instantia-
tion `FortranExpression([X],[M],MachineFloat)` is the domain of expressions
containing the scalar X and the array M.

This allows us to create expressions like:

```
f :  FortranExpression([X],[M],MachineFloat) := sin(X)+M[3,1]
```

but not

```
f :  FortranExpression([X],[M],MachineFloat) := sin(M)+Y
```

Those ASPs which represent expressions usually export a **coerce** from
an appropriate instantiation of `FortranExpression` (or perhaps `Vector`
`FortranExpression` etc.). For convenience there are also retractions
from appropriate instantiations of `Expression`, `Polynomial` and `Fraction`
`Polynomial`.

Providing ASPs via `FortranCode`

`FortranCode` allows us to build arbitrarily complex ASPs via a kind of pseudo-
code. It is described fully in **??** on page **??**.

Every ASP exports two **coerce** functions: one from `FortranCode`
and one from `List FortranCode`. There is also a **coerce** from
`Record( localSymbols: SymbolTable, code: List FortranCode)` which
is used for passing extra symbol information about the ASP.

So for example, to integrate the function abs(x) we could use the built-in **abs**
function. But suppose we want to get back to basics and define it directly, then
we could do the following:

```
d01ajf(-1.0, 1.0, 0.0, 1.0e-5, 800, 200, -1, cond(LT(X,0),
assign(F,-X), assign(F,X))) result
```

The **cond** operation creates a conditional clause and the **assign** an assignment
statement.

Providing ASPs via `FileName`

Suppose we have created the file "asp.f" as follows:

```
      DOUBLE PRECISION FUNCTION F(X)
      DOUBLE PRECISION X
      F=4.0D0/(X*X+1.0D0)
      RETURN
      END
```

and wish to pass it to the NAG routine **d01ajf** which performs one-dimensional quadrature. We can do this as follows:

```
d01ajf(0.0 ,1.0, 0.0, 1.0e-5, 800, 200, -1, "asp.f")
```

15.3.4 General Fortran-generation utilities in Axiom

This section describes more advanced facilities which are available to users who wish to generate Fortran code from within Axiom. There are facilities to manipulate templates, store type information, and generate code fragments or complete programs.

Template Manipulation

A template is a skeletal program which is "fleshed out" with data when it is processed. It is a sequence of *active* and *passive* parts: active parts are sequences of Axiom commands which are processed as if they had been typed into the interpreter; passive parts are simply echoed verbatim on the Fortran output stream.

Suppose, for example, that we have the following template, stored in the file "test.tem":

```
-- A simple template
beginVerbatim
      DOUBLE PRECISION FUNCTION F(X)
      DOUBLE PRECISION X
endVerbatim
outputAsFortran("F",f)
beginVerbatim
      RETURN
      END
endVerbatim
```

The passive parts lie between the two tokens `beginVerbatim` and `endVerbatim`. There are two active statements: one which is simply an Axiom (`--`) comment, and one which produces an assignment to the current value of `f`. We could use it as follows:

```
(4) ->f := 4.0/(1+X**2)
```

```
          4
   (4)   ------
          2
         X  + 1
```

```
(5) ->processTemplate "test.tem"
      DOUBLE PRECISION FUNCTION F(X)
      DOUBLE PRECISION X
      F=4.0D0/(X*X+1.0D0)
      RETURN
      END
```

```
   (5)  "CONSOLE"
```

(A more reliable method of specifying the filename will be introduced below.)
Note that the Fortran assignment `F=4.0D0/(X*X+1.0D0)` automatically con-
verted 4.0 and 1 into DOUBLE PRECISION numbers; in general, the Axiom
Fortran generation facility will convert anything which should be a floating point
object into either a Fortran REAL or DOUBLE PRECISION object.

Which alternative is used is determined by the command

```
)set fortran precision
```

It is sometimes useful to end a template before the file itself ends (e.g. to allow
the template to be tested incrementally or so that a piece of text describing how
the template works can be included). It is of course possible to "comment-out"
the remainder of the file. Alternatively, the single token `endInput` as part of an
active portion of the template will cause processing to be ended prematurely at
that point.

The **processTemplate** command comes in two flavours. In the first case, il-
lustrated above, it takes one argument of domain `FileName`, the name of the
template to be processed, and writes its output on the current Fortran out-
put stream. In general, a filename can be generated from *directory*, *name* and
extension components, using the operation **filename**, as in

```
processTemplate filename("","test","tem")
```

There is an alternative version of **processTemplate**, which takes two argu-
ments (both of domain `FileName`). In this case the first argument is the name
of the template to be processed, and the second is the file in which to write the
results. Both versions return the location of the generated Fortran code as their
result ("`CONSOLE`" in the above example).

It is sometimes useful to be able to mix active and passive parts of a line or state-
ment. For example you might want to generate a Fortran Comment describing

your data set. For this kind of application we provide three functions as follows:

fortranLiteral	writes a string on the Fortran output stream
fortranCarriageReturn	writes a carriage return on the Fortran output stream
fortranLiteralLine	writes a string followed by a return on the Fortran output stream

So we could create our comment as follows:

```
m := matrix [ [1,2,3],[4,5,6] ]
```

```
fortranLiteralLine concat ["C     The Matrix has ",
nrows(m)::String, " rows and ", ncols(m)::String, " columns"]
```

or, alternatively:

```
fortranLiteral "C     The Matrix has "
```

```
fortranLiteral(nrows(m)::String)
```

```
fortranLiteral " rows and "
```

```
fortranLiteral(ncols(m)::String)
```

```
fortranLiteral " columns"
```

```
fortranCarriageReturn()
```

We should stress that these functions, together with the **outputAsFortran** function are the *only* sure ways of getting output to appear on the Fortran output stream. Attempts to use Axiom commands such as **output** or **writeline** may appear to give the required result when displayed on the console, but will give the wrong result when Fortran and algebraic output are sent to differing locations. On the other hand, these functions can be used to send helpful messages to the user, without interfering with the generated Fortran.

Manipulating the Fortran Output Stream

Sometimes it is useful to manipulate the Fortran output stream in a program, possibly without being aware of its current value. The main use of this is for gathering type declarations (see "Fortran Types" below) but it can be useful in other contexts as well. Thus we provide a set of commands to manipulate a stack of (open) output streams. Only one stream can be written to at any given time. The stack is never empty—its initial value is the console or the current value of the Fortran output stream, and can be determined using

```
topFortranOutputStack()
```

(see below). The commands available to manipulate the stack are:

clearFortranOutputStack   resets the stack to the console

pushFortranOutputStack   pushes a `FileName` onto the stack

popFortranOutputStack   pops the stack

showFortranOutputStack   returns the current stack

topFortranOutputStack   returns the top element of the stack

These commands are all part of `FortranOutputStackPackage`.

Fortran Types

When generating code it is important to keep track of the Fortran types of the objects which we are generating. This is useful for a number of reasons, not least to ensure that we are actually generating legal Fortran code. The current type system is built up in several layers, and we shall describe each in turn.

FortranScalarType

This domain represents the simple Fortran datatypes: REAL, DOUBLE PRE-CISION, COMPLEX, LOGICAL, INTEGER, and CHARACTER. It is possible to **coerce** a `String` or `Symbol` into the domain, test whether two objects are equal, and also apply the predicate functions **real?** etc.

FortranType

This domain represents "full" types: i.e., datatype plus array dimensions (where appropriate) plus whether or not the parameter is an external subprogram. It is possible to **coerce** an object of `FortranScalarType` into the domain or

construct one from an element of `FortranScalarType`, a list of `Polynomial`
`Integers` (which can of course be simple integers or symbols) representing its
dimensions, and a `Boolean` declaring whether it is external or not. The list of di-
mensions must be empty if the `Boolean` is `true`. The functions **scalarTypeOf**,
dimensionsOf and **external?** return the appropriate parts, and it is possible
to get the various basic Fortran Types via functions like **fortranReal**.

For example:

```
type:=construct(real,[i,10],false)$FortranType
```

or

```
type:=[real,[i,10],false]$FortranType
```

```
scalarTypeOf type
```

```
dimensionsOf type
```

```
external?  type
```

```
fortranLogical()
```

```
construct(integer,[],true)$FortranType
```

SymbolTable

This domain creates and manipulates a symbol table for generated Fortran
code. This is used by `FortranProgram` to represent the types of objects in a
subprogram. The commands available are:

empty	creates a new `SymbolTable`
declare	creates a new entry in a table
fortranTypeOf	returns the type of an object in a table
parametersOf	returns a list of all the symbols in the table
typeList	returns a list of all objects of a given type
typeLists	returns a list of lists of all objects sorted by type
externalList	returns a list of all `EXTERNAL` objects
printTypes	produces Fortran type declarations from a table

```
symbols := empty()$SymbolTable
```

```
declare!(X,fortranReal(),symbols)
```

```
declare!(M,construct(real,[i,j],false)$FortranType,symbols)
```

```
declare!([i,j],fortranInteger(),symbols)
```

```
symbols
```

```
fortranTypeOf(i,symbols)
```

```
typeList(real,symbols)
```

```
printTypes symbols
```

TheSymbolTable

This domain creates and manipulates one global symbol table to be used, for example, during template processing. It is also used when linking to external Fortran routines. The information stored for each subprogram (and the main program segment, where relevant) is:

- its name;

- its return type;

- its argument list;

- and its argument types.

Initially, any information provided is deemed to be for the main program segment.

Issuing the following command indicates that from now on all information refers to the subprogram F.

```
newSubProgram F
```

It is possible to return to processing the main program segment by issuing the command:

```
endSubProgram()
```

The following commands exist:

returnType	declares the return type of the current subprogram
returnTypeOf	returns the return type of a subprogram
argumentList	declares the argument list of the current subprogram
argumentListOf	returns the argument list of a subprogram
declare	provides type declarations for parameters of the current subprogram
symbolTableOf	returns the symbol table of a subprogram
printHeader	produces the Fortran header for the current subprogram

In addition there are versions of these commands which are parameterised by the name of a subprogram, and others parameterised by both the name of a subprogram and by an instance of `TheSymbolTable`.

```
newSubProgram F
```

```
argumentList!(F,[X])
```

```
returnType!(F,real)
```

```
declare!(X,fortranReal(),F)
```

```
printHeader F
```

Advanced Fortran Code Generation

This section describes facilities for representing Fortran statements, and building up complete subprograms from them.

Switch

This domain is used to represent statements like `x < y`. Although these can be represented directly in Axiom, it is a little cumbersome, since Axiom evaluates the last statement, for example, to `true` (since x is lexicographically less than y).

Instead we have a set of operations, such as **LT** to represent $<$, to let us build such statements. The available constructors are:

LT	$<$
GT	$>$
LE	$\leq$
GE	$\geq$
EQ	$=$
AND	`and`
OR	`or`
NOT	`not`

So for example:

```
LT(x,y)
```

FortranCode

This domain represents code segments or operations: currently assignments, conditionals, blocks, comments, gotos, continues, various kinds of loops, and return statements.

For example we can create quite a complicated conditional statement using assignments, and then turn it into Fortran code:

```
c :=
cond(LT(X,Y),assign(F,X),cond(GT(Y,Z),assign(F,Y),assign(F,Z)))
```

```
printCode c
```

The Fortran code is printed on the current Fortran output stream.

FortranProgram

This domain is used to construct complete Fortran subprograms out of elements of `FortranCode`. It is parameterised by the name of the target subprogram (a

Symbol), its return type (from Union(FortranScalarType,"void")), its argu-
ments (from List Symbol), and its symbol table (from SymbolTable). One can
coerce elements of either FortranCode or Expression into it.

First of all we create a symbol table:

```
symbols := empty()$SymbolTable
```

Now put some type declarations into it:

```
declare!([X,Y],fortranReal(),symbols)
```

Then (for convenience) we set up the particular instantiation of FortranProgram

```
FP := FortranProgram(F,real,[X,Y],symbols)
```

Create an object of type Expression(Integer):

```
asp := X*sin(Y)
```

Now **coerce** it into FP, and print its Fortran form:

```
outputAsFortran(asp::FP)
```

We can generate a FortranProgram using *FortranCode*. For example:
Augment our symbol table:

```
declare!(Z,fortranReal(),symbols)
```

and transform the conditional expression we prepared earlier:

```
outputAsFortran([c,returns()]::FP)
```

15.3.5   Some technical information

The model adopted for the link is a server-client configuration – Axiom act-
ing as a client via a local agent (a process called **nagman**). The server side is
implemented by the **nagd** daemon process which may run on a different host.
The **nagman** local agent is started by default whenever you start Axiom. The
nagd server must be started separately. Instructions for installing and running
the server are supplied in **??** on page **??**. Use the `)set naglink host` system
command to point your local agent to a server in your network.

On the Axiom side, one sees a set of *packages* (ask Browse for *Nag**) for each chapter, each exporting operations with the same name as a routine in the Nag Library. The arguments and return value of each operation belong to standard Axiom types.

The `man` pages for the Nag Library are accessible via the description of each operation in Browse (among other places).

In the implementation of each operation, the set of inputs is passed to the local agent `nagman`, which makes a Remote Procedure Call (RPC) to the remote `nagd` daemon process. The local agent receives the RPC results and forwards them to the Axiom workspace where they are interpreted appropriately.

How are Fortran subroutines turned into RPC calls? For each Fortran routine in the Nag Library, a C main() routine is supplied. Its job is to assemble the RPC input (numeric) data stream into the appropriate Fortran data structures for the routine, call the Fortran routine from C and serialize the results into an RPC output data stream.

Many Nag Library routines accept ASPs (Argument Subprogram Parameters). These specify user-supplied Fortran routines (e.g. a routine to supply values of a function is required for numerical integration). How are they handled? There are new facilities in Axiom to help. A set of Axiom domains has been provided to turn values in standard Axiom types (such as Expression Integer) into the appropriate piece of Fortran for each case (a filename pointing to Fortran source for the ASP can always be supplied instead). Ask Browse for *Asp** to see these domains. The Fortran fragments are included in the outgoing RPC stream, but `nagd` intercepts them, compiles them, and links them with the main() C program before executing the resulting program on the numeric part of the RPC stream.

15.4   Interactive Front-end and Language

The `leave` keyword has been replaced by the `break` keyword for compatibility with the new Axiom extension language. See section 5.4.3 on page 207 for more information.

Curly braces are no longer used to create sets. Instead, use **set** followed by a bracketed expression. For example,

```
set [1,2,3,4]
```

Curly braces are now used to enclose a block (see section 5.2 on page 199 for more information). For compatibility, a block can still be enclosed by parentheses as well.

"Free functions" created by the Aldor compiler can now be loaded and used within the Axiom interpreter. A *free function* is a library function that is implemented outside a domain or category constructor.

New coercions to and from type `Expression` have been added. For example, it is now possible to map a polynomial represented as an expression to an appropriate polynomial type.

Various messages have been added or rewritten for clarity.

15.5   Library

The `FullPartialFractionExpansion` domain has been added. This domain computes factor-free full partial fraction expansions. See section FullPartial-FractionExpansion for examples.

We have implemented the Bertrand/Cantor algorithm for integrals of hyper-elliptic functions. This brings a major speedup for some classes of algebraic integrals.

We have implemented a new (direct) algorithm for integrating trigonometric functions. This brings a speedup and an improvement in the answer quality.

The `SmallFloat` domain has been renamed `DoubleFloat` and `SmallInteger` has been renamed `SingleInteger`. The new abbreviations as `DFLOAT` and `SINT`, respectively. We have defined the macro `SF`, the old abbreviation for `SmallFloat`, to expand to `DoubleFloat` and modified the documentation and input file examples to use the new names and abbreviations. You should do the same in any private Axiom files you have.

There are many new categories, domains and packages related to the NAG Library Link facility. See the file

src/algebra/exposed.lsp

for a list of constructors in the **naglink** Axiom exposure group.

We have made improvements to the differential equation solvers and there is a new facility for solving systems of first-order linear differential equations. In particular, an important fix was made to the solver for inhomogeneous linear ordinary differential equations that corrected the calculation of particular solutions. We also made improvements to the polynomial and transcendental equation solvers including the ability to solve some classes of systems of transcendental equations.

The efficiency of power series have been improved and left and right expansions of $tan(f(x))$ at $x = a$ pole of $f(x)$ can now be computed. A number of power series bugs were fixed and the `GeneralUnivariatePowerSeries` domain was added. The power series variable can appear in the coefficients and when this happens, you cannot differentiate or integrate the series. Differentiation and integration with respect to other variables is supported.

A domain was added for representing asymptotic expansions of a function at an exponential singularity.

For limits, the main new feature is the exponential expansion domain used to treat certain exponential singularities. Previously, such singularities were treated in an *ad hoc* way and only a few cases were covered. Now Axiom can do things like

```
limit( (x+1)**(x+1)/x**x - x**x/(x-1)**(x-1), x = %plusInfinity)
```

in a systematic way. It only does one level of nesting, though. In other words, we can handle $exp(somefunctionwithapole)$, but not $exp(exp(somefunctionwithapole))$.

The computation of integral bases has been improved through careful use of Hermite row reduction. A P-adic algorithm for function fields of algebraic curves in finite characteristic has also been developed.

Miscellaneous: There is improved conversion of definite and indefinite integrals to `InputForm`; binomial coefficients are displayed in a new way; some new simplifications of radicals have been implemented; the operation **complexForm** for converting to rectangular coordinates has been added; symmetric product operations have been added to `LinearOrdinaryDifferentialOperator`.

15.6 HyperTex

The buttons on the titlebar and scrollbar have been replaced with ones which have a 3D effect. You can change the foreground and background colors of these "controls" by including and modifying the following lines in your **.Xdefaults** file.

```
Axiom.hyperdoc.ControlBackground: White
Axiom.hyperdoc.ControlForeground: Black
```

For various reasons, HyperDoc sometimes displays a secondary window. You can control the size and placement of this window by including and modifying the following line in your **.Xdefaults** file.

```
Axiom.hyperdoc.FormGeometry: =950x450+100+0
```

This setting is a standard X Window System geometry specification: you are requesting a window 950 pixels wide by 450 deep and placed in the upper left corner.

Some key definitions have been changed to conform more closely with the CUA guidelines. Press F9 to see the current definitions.

Input boxes (for example, in the Browser) now accept paste-ins from the X Window System. Use the second button to paste in something you have previously copied or cut. An example of how you can use this is that you can paste the type from an Axiom computation into the main Browser input box.

15.7 Documentation

We describe here a few additions to the on-line version of the AXIOM book which you can read with HyperDoc.

A section has been added to the graphics chapter, describing how to build two-dimensional graphs from lists of points. An example is given showing how to read the points from a file. See section 7.1.9 on page 310 for details.

A further section has been added to that same chapter, describing how to add a two-dimensional graph to a viewport which already contains other graphs. See section 7.1.10 on page 317 for details.

Chapter 3 and the on-line HyperDoc help have been unified.

An explanation of operation names ending in "?" and "!" has been added to the first chapter. See the end of the section 1.3.6 on page 80 for details.

An expanded explanation of using predicates has been added to the sixth chapter. See the example involving **evenRule** in the middle of the section 6.21 on page 288 for details.

Documentation for the `)compile`, `)library` and `)load` commands has been greatly changed. This reflects the ability of the `)compile` to now invoke the Aldor compiler, the impending deletion of the `)load` command and the new `)library` command. The `)library` command replaces `)load` and is compatible with the compiled output from both the old and new compilers.

Chapter 1

Axiom System Commands

This chapter describes system commands, the command-line facilities used to control the Axiom environment. The first section is an introduction and discusses the common syntax of the commands available.

1.1 Introduction

System commands are used to perform Axiom environment management. Among the commands are those that display what has been defined or computed, set up multiple logical Axiom environments (frames), clear definitions, read files of expressions and commands, show what functions are available, and terminate Axiom.

Some commands are restricted: the commands

```
)set userlevel interpreter
)set userlevel compiler
)set userlevel development
```

set the user-access level to the three possible choices. All commands are available at `development` level and the fewest are available at `interpreter` level. The default user-level is `interpreter`. In addition to the `)set` command (discussed in **??** on page **??**) you can use the HyperDoc settings facility to change the *user-level.*

Each command listing begins with one or more syntax pattern descriptions plus examples of related commands. The syntax descriptions are intended to be easy to read and do not necessarily represent the most compact way of specifying all possible arguments and options; the descriptions may occasionally be redundant.

All system commands begin with a right parenthesis which should be in the first available column of the input line (that is, immediately after the input prompt,

if any). System commands may be issued directly to Axiom or be included in
.input files.

A system command *argument* is a word that directly follows the command name
and is not followed or preceded by a right parenthesis. A system command *option* follows the system command and is directly preceded by a right parenthesis.
Options may have arguments: they directly follow the option. This example may
make it easier to remember what is an option and what is an argument:

> `)syscmd` *arg1 arg2* `)opt1` *opt1arg1 opt1arg2* `)opt2` *opt2arg1* ...

In the system command descriptions, optional arguments and options are en-
closed in brackets ("`[`" and "`]`"). If an argument or option name is in italics,
it is meant to be a variable and must have some actual value substituted for
it when the system command call is made. For example, the syntax pattern
description

`)read` *fileName* `[)quietly]`

would imply that you must provide an actual file name for *fileName* but need
not use the `)quietly` option. Thus

`)read matrix.input`

is a valid instance of the above pattern.

System command names and options may be abbreviated and may be in upper
or lower case. The case of actual arguments may be significant, depending on
the particular situation (such as in file names). System command names and
options may be abbreviated to the minimum number of starting letters so that
the name or option is unique. Thus

`)s Integer`

is not a valid abbreviation for the `)set` command, because both `)set` and
`)show` begin with the letter "s". Typically, two or three letters are sufficient for
disambiguating names. In our descriptions of the commands, we have used no
abbreviations for either command names or options.

In some syntax descriptions we use a vertical line "|" to indicate that you must
specify one of the listed choices. For example, in

`)set output fortran on | off`

only `on` and `off` are acceptable words for following `boot`. We also sometimes
use "..." to indicate that additional arguments or options of the listed form
are allowed. Finally, in the syntax descriptions we may also list the syntax of
related commands.

1.2   )abbreviation

User Level Required: compiler

Command Syntax:

>)abbreviation query [*nameOrAbbrev*]
>
>)abbreviation category *abbrev fullname* [)quiet]
>
>)abbreviation domain *abbrev fullname* [)quiet]
>
>)abbreviation package *abbrev fullname* [)quiet]
>
>)abbreviation remove *nameOrAbbrev*

Command Description:

This command is used to query, set and remove abbreviations for category, domain and package constructors. Every constructor must have a unique abbreviation. This abbreviation is part of the name of the subdirectory under which the components of the compiled constructor are stored. Furthermore, by issuing this command you let the system know what file to load automatically if you use a new constructor. Abbreviations must start with a letter and then be followed by up to seven letters or digits. Any letters appearing in the abbreviation must be in uppercase.

When used with the `query` argument, this command may be used to list the name associated with a particular abbreviation or the abbreviation for a constructor. If no abbreviation or name is given, the names and corresponding abbreviations for *all* constructors are listed.

The following shows the abbreviation for the constructor `List`:

```
)abbreviation query List
```

The following shows the constructor name corresponding to the abbreviation `NNI`:

```
)abbreviation query NNI
```

The following lists all constructor names and their abbreviations.

```
)abbreviation query
```

To add an abbreviation for a constructor, use this command with `category`, `domain` or `package`. The following add abbreviations to the system for a category, domain and package, respectively:

```
)abbreviation domain   SET Set
)abbreviation category COMPCAT  ComplexCategory
)abbreviation package  LIST2MAP ListToMap
```

If the)`quiet` option is used, no output is displayed from this command. You would normally only define an abbreviation in a library source file. If this command is issued for a constructor that has already been loaded, the constructor will be reloaded next time it is referenced. In particular, you can use this command to force the automatic reloading of constructors.

To remove an abbreviation, the `remove` argument is used. This is usually only used to correct a previous command that set an abbreviation for a constructor name. If, in fact, the abbreviation does exist, you are prompted for confirmation of the removal request. Either of the following commands will remove the abbreviation `VECTOR2` and the constructor name `VectorFunctions2` from the system:

```
)abbreviation remove VECTOR2
)abbreviation remove VectorFunctions2
```

Also See:)`compile`

1.3   )boot

User Level Required: development

Command Syntax:

>   )`boot` *bootExpression*

Command Description:

This command is used by Axiom system developers to execute expressions written in the BOOT language. For example,

```
)boot times3(x) == 3*x
```

creates and compiles the Common Lisp function "times3" obtained by translating the BOOT code.

Also See:)`fin` ,)`lisp` ,)`set` , and)`system` .

1.4   )cd

User Level Required: interpreter

Command Syntax:

>   )`cd` *directory*

Command Description:

This command sets the Axiom working current directory. The current directory is used for looking for input files (for)`read`), Axiom library source files (for

`)compile`), saved history environment files (for `)history )restore`), compiled Axiom library files (for `)library`), and files to edit (for `)edit`). It is also used for writing spool files (via `)spool`), writing history input files (via `)history )write`) and history environment files (via `)history )save`),and compiled Axiom library files (via `)compile`).

If issued with no argument, this command sets the Axiom current directory to your home directory. If an argument is used, it must be a valid directory name. Except for the ")" at the beginning of the command, this has the same syntax as the operating system `cd` command.

Also See: `)compile` , `)edit` , `)history` , `)library` , `)read` , and `)spool` .

1.5   )close

User Level Required: interpreter

Command Syntax:

```
)close
```

```
)close )quietly
```

Command Description:

This command is used to close down interpreter client processes. Such processes are started by HyperDoc to run Axiom examples when you click on their text. When you have finished examining or modifying the example and you do not want the extra window around anymore, issue

```
)close
```

to the Axiom prompt in the window.

If you try to close down the last remaining interpreter client process, Axiom will offer to close down the entire Axiom session and return you to the operating system by displaying something like

```
This is the last AXIOM session. Do you want to kill AXIOM?
```

Type "y" (followed by the Return key) if this is what you had in mind. Type "n" (followed by the Return key) to cancel the command.

You can use the `)quietly` option to force Axiom to close down the interpreter client process without closing down the entire Axiom session.

Also See: `)quit` and `)pquit` .

1.6   )clear

User Level Required: interpreter

Command Syntax:

)clear all

)clear completely

)clear properties all

)clear properties *obj1* [*obj2* ...]

)clear value all

)clear value *obj1* [*obj2* ...]

)clear mode all

)clear mode *obj1* [*obj2* ...]

Command Description:

This command is used to remove function and variable declarations, definitions and values from the workspace. To empty the entire workspace and reset the step counter to 1, issue

```
)clear all
```

To remove everything in the workspace but not reset the step counter, issue

```
)clear properties all
```

To remove everything about the object **x**, issue

```
)clear properties x
```

To remove everything about the objects **x, y** and **f**, issue

```
)clear properties x y f
```

The word `properties` may be abbreviated to the single letter "**p**".

```
)clear p all
)clear p x
)clear p x y f
```

All definitions of functions and values of variables may be removed by either

```
)clear value all
)clear v all
```

This retains whatever declarations the objects had. To remove definitions and values for the specific objects `x, y` and `f`, issue

```
)clear value x y f
)clear v x y f
```

To remove the declarations of everything while leaving the definitions and values, issue

```
)clear mode  all
)clear m all
```

To remove declarations for the specific objects `x, y` and `f`, issue

```
)clear mode x y f
)clear m x y f
```

The `)display names` and `)display properties` commands may be used to see what is currently in the workspace.

The command

```
)clear completely
```

does everything that `)clear all` does, and also clears the internal system function and constructor caches.

Also See: `)display` , `)history` , and `)undo` .

1.7   )compile

User Level Required: compiler

Command Syntax:

> `)compile`
>
> `)compile` *fileName*
>
> `)compile` *fileName*`.as`
>
> `)compile` *directory/fileName*`.as`
>
> `)compile` *fileName*`.ao`
>
> `)compile` *directory/fileName*`.ao`

)compile *fileName*.al

)compile *directory/fileName*.al

)compile *fileName*.lsp

)compile *directory/fileName*.lsp

)compile *fileName*.spad

)compile *directory/fileName*.spad

)compile *fileName*)new

)compile *fileName*)old

)compile *fileName*)translate

)compile *fileName*)quiet

)compile *fileName*)noquiet

)compile *fileName*)moreargs

)compile *fileName*)onlyargs

)compile *fileName*)break

)compile *fileName*)nobreak

)compile *fileName*)library

)compile *fileName*)nolibrary

)compile *fileName*)vartrace

)compile *fileName*)constructor *nameOrAbbrev*

Command Description:

You use this command to invoke the new Axiom library compiler or the old Axiom system compiler. The)compile system command is actually a combination of Axiom processing and a call to the Aldor compiler. It is performing double-duty, acting as a front-end to both the Aldor compiler and the old Axiom system compiler. (The old Axiom system compiler was written in Lisp and was an integral part of the Axiom environment. The Aldor compiler is written in C and executed by the operating system when called from within Axiom.)

The command compiles files with file extensions *.as, .ao* and *.al* with the Aldor compiler and files with file extension *.spad* with the old Axiom system compiler. It also can compile files with file extension *.lsp*. These are assumed to be Lisp files genererated by the Aldor compiler. If you omit the file extension, the command looks to see if you have specified the)new or)old option. If you have given one of these options, the corresponding compiler is used. Otherwise, the

command first looks in the standard system directories for files with extension *.as, .ao* and *.al* and then files with extension *.spad*. The first file found has the appropriate compiler invoked on it. If the command cannot find a matching file, an error message is displayed and the command terminates.

The `)translate` option is used to invoke a special version of the old system compiler that will translate a *.spad* file to a *.as* file. That is, the *.spad* file will be parsed and analyzed and a file using the new syntax will be created. By default, the *.as* file is created in the same directory as the *.spad* file. If that directory is not writable, the current directory is used. If the current directory is not writable, an error message is given and the command terminates. Note that `)translate` implies the `)old` option so the file extension can safely be omitted. If `)translate` is given, all other options are ignored. Please be aware that the translation is not necessarily one hundred percent complete or correct. You should attempt to compile the output with the Aldor compiler and make any necessary corrections.

We now describe the options for the new Aldor compiler.

The first thing `)compile` does is look for a source code filename among its arguments. Thus

```
)compile mycode.as
)compile /u/jones/as/mycode.as
)compile mycode
```

all invoke `)compiler` on the file `/u/jones/as/mycode.as` if the current Axiom working directory is `/u/jones/as.` (Recall that you can set the working directory via the `)cd` command. If you don't set it explicitly, it is the directory from which you started Axiom.)

This is frequently all you need to compile your file. This simple command:

1. Invokes the Aldor compiler and produces Lisp output.

2. Calls the Lisp compiler if the Aldor compilation was successful.

3. Uses the `)library` command to tell Axiom about the contents of your compiled file and arrange to have those contents loaded on demand.

Should you not want the `)library` command automatically invoked, call `)compile` with the `)nolibrary` option. For example,

```
)compile mycode.as )nolibrary
```

The general description of Aldor command line arguments is in the Aldor documentation. The default options used by the `)compile` command can be viewed and set using the `)set compiler args` Axiom system command. The current defaults are

```
-O -Fasy -Fao -Flsp -laxiom -Mno-AXL_W_WillObsolete -DAxiom
```

These options mean:

- `-O`: perform all optimizations,

- `-Fasy`: generate a `.asy` file,

- `-Fao`: generate a `.ao` file,

- `-Flsp`: generate a `.lsp` (Lisp) file,

- `-laxiom`: use the `axiom` library `libaxiom.al`,

- `-Mno-AXL_W_WillObsolete`: do not display messages about older generated files becoming obsolete, and

- `-DAxiom`: define the global assertion `Axiom` so that the Aldor libraries for generating stand-alone code are not accidentally used with Axiom.

To supplement these default arguments, use the `)moreargs` option on `)compile`. For example,

```
)compile mycode.as )moreargs "-v"
```

uses the default arguments and appends the `-v` (verbose) argument flag. The additional argument specification **must be enclosed in double quotes.**

To completely replace these default arguments for a particular use of `)compile`, use the `)onlyargs` option. For example,

```
)compile mycode.as )onlyargs "-v -O"
```

only uses the `-v` (verbose) and `-O` (optimize) arguments. The argument specification **must be enclosed in double quotes.** In this example, Lisp code is not produced and so the compilation output will not be available to Axiom.

To completely replace the default arguments for all calls to `)compile` within your Axiom session, use `)set compiler args`. For example, to use the above arguments for all compilations, issue

```
)set compiler args "-v -O"
```

Make sure you include the necessary `-l` and `-Y` arguments along with those needed for Lisp file creation. As above, **the argument specification must be enclosed in double quotes.**

By default, the `)library` system command *exposes* all domains and categories it processes. This means that the Axiom intepreter will consider those domains and categories when it is trying to resolve a reference to a function. Sometimes

domains and categories should not be exposed. For example, a domain may just be used privately by another domain and may not be meant for top-level use. The `)library` command should still be used, though, so that the code will be loaded on demand. In this case, you should use the `)nolibrary` option on `)compile` and the `)noexpose` option in the `)library` command. For example,

```
)compile mycode.as )nolibrary
)library mycode )noexpose
```

Once you have established your own collection of compiled code, you may find it handy to use the `)dir` option on the `)library` command. This causes `)library` to process all compiled code in the specified directory. For example,

```
)library )dir /u/jones/as/quantum
```

You must give an explicit directory after `)dir`, even if you want all compiled code in the current working directory processed, e.g.

```
)library )dir .
```

The `)compile` command works with several file extensions. We saw above what happens when it is invoked on a file with extension `.as`. A `.ao` file is a portable binary compiled version of a `.as` file, and `)compile` simply passes the `.ao` file onto Aldor. The generated Lisp file is compiled and `)library` is automatically called, just as if you had specified a `.as` file.

A `.al` file is an archive file containing `.ao` files. The archive is created (on Unix systems) with the `ar` program. When `)compile` is given a `.al` file, it creates a directory whose name is based on that of the archive. For example, if you issue

```
)compile mylib.al
```

the directory `mylib.axldir` is created. All members of the archive are unarchived into the directory and `)compile` is called on each `.ao` file found. It is your responsibility to remove the directory and its contents, if you choose to do so.

A `.lsp` file is a Lisp source file, presumably, in our context, generated by Aldor when called with the `-Flsp` option. When `)compile` is used with a `.lsp` file, the Lisp file is compiled and `)library` is called. You must also have present a `.asy` generated from the same source file.

The following are descriptions of options for the old system compiler.

You can compile category, domain, and package constructors contained in files with file extension *.spad*. You can compile individual constructors or every constructor in a file.

The full filename is remembered between invocations of this command and `)edit` commands. The sequence of commands

```
)compile matrix.spad
)edit
)compile
```

will call the compiler, edit, and then call the compiler again on the file **matrix.spad.** If you do not specify a *directory,* the working current directory (see **??** on page **??**) is searched for the file. If the file is not found, the standard system directories are searched.

If you do not give any options, all constructors within a file are compiled. Each constructor should have an `)abbreviation` command in the file in which it is defined. We suggest that you place the `)abbreviation` commands at the top of the file in the order in which the constructors are defined. The list of commands serves as a table of contents for the file.

The `)library` option causes directories containing the compiled code for each constructor to be created in the working current directory. The name of such a directory consists of the constructor abbreviation and the **.NRLIB** file extension. For example, the directory containing the compiled code for the `MATRIX` constructor is called **MATRIX.NRLIB.** The `)nolibrary` option says that such files should not be created. The default is `)library`. Note that the semantics of `)library` and `)nolibrary` for the new Aldor compiler and for the old system compiler are completely different.

The `)vartrace` option causes the compiler to generate extra code for the constructor to support conditional tracing of variable assignments. (see **??** on page **??**). Without this option, this code is suppressed and one cannot use the `)vars` option for the trace command.

The `)constructor` option is used to specify a particular constructor to compile. All other constructors in the file are ignored. The constructor name or abbreviation follows `)constructor`. Thus either

```
)compile matrix.spad )constructor RectangularMatrix
```

or

```
)compile matrix.spad )constructor RMATRIX
```

compiles the `RectangularMatrix` constructor defined in **matrix.spad.**

The `)break` and `)nobreak` options determine what the old system compiler does when it encounters an error. `)break` is the default and it indicates that processing should stop at the first error. The value of the `)set break` variable then controls what happens.

Also See: `)abbreviation` , `)edit` , and `)library` .

1.8)display

User Level Required: interpreter

Command Syntax:

)display all

)display properties

)display properties all

)display properties [*obj1* [*obj2* ...]]

)display value all

)display value [*obj1* [*obj2* ...]]

)display mode all

)display mode [*obj1* [*obj2* ...]]

)display names

)display operations *opName*

Command Description:

This command is used to display the contents of the workspace and signatures of functions with a given name.[1]

The command

```
)display names
```

lists the names of all user-defined objects in the workspace. This is useful if you do not wish to see everything about the objects and need only be reminded of their names.

The commands

```
)display all
)display properties
)display properties all
```

all do the same thing: show the values and types and declared modes of all variables in the workspace. If you have defined functions, their signatures and definitions will also be displayed.

To show all information about a particular variable or user functions, for example, something named `d`, issue

---

[1]A *signature* gives the argument and return types of a function.

```
)display properties d
```

To just show the value (and the type) of **d**, issue

```
)display value d
```

To just show the declared mode of **d**, issue

```
)display mode d
```

All modemaps for a given operation may be displayed by using `)display` `operations`. A *modemap* is a collection of information about a particular reference to an operation. This includes the types of the arguments and the return value, the location of the implementation and any conditions on the types. The modemap may contain patterns. The following displays the modemaps for the operation **complex**:

```
)d op complex
```

Also See: `)clear` , `)history` , `)set` , `)show` , and `)what` .

1.9   )edit

User Level Required: interpreter

Command Syntax:

> `)edit` [*filename*]

Command Description:

This command is used to edit files. It works in conjunction with the `)read` and `)compile` commands to remember the name of the file on which you are working. By specifying the name fully, you can edit any file you wish. Thus

```
)edit /u/julius/matrix.input
```

will place you in an editor looking at the file **/u/julius/matrix.input**. By default, the editor is **vi**, but if you have an EDITOR shell environment variable defined, that editor will be used. When Axiom is running under the X Window System, it will try to open a separate **xterm** running your editor if it thinks one is necessary. For example, under the Korn shell, if you issue

```
export EDITOR=emacs
```

then the emacs editor will be used by `)edit`.

If you do not specify a file name, the last file you edited, read or compiled will be used. If there is no "last file" you will be placed in the editor editing an empty unnamed file.

It is possible to use the `)system` command to edit a file directly. For example,

```
)system emacs /etc/rc.tcpip
```

calls `emacs` to edit the file.

Also See: `)system` , `)compile` , and `)read` .

1.10   )fin

User Level Required: development

Command Syntax:

>   `)fin`

Command Description:

This command is used by Axiom developers to leave the Axiom system and return to the underlying Common Lisp system. To return to Axiom, issue the "(|spad|)" function call to Common Lisp.

Also See: `)pquit` and `)quit` .

1.11   )frame

User Level Required: interpreter

Command Syntax:

>   `)frame new` *frameName*

>   `)frame drop` [*frameName*]

>   `)frame next`

>   `)frame last`

>   `)frame names`

>   `)frame import` *frameName* [*objectName1* [*objectName2* ...]]

>   `)set message frame on | off`

>   `)set message prompt frame`

Command Description:

A *frame* can be thought of as a logical session within the physical session that you get when you start the system. You can have as many frames as you want, within the limits of your computer's storage, paging space, and so on. Each frame has its own *step number*, *environment* and *history.* You can have a variable named `a` in one frame and it will have nothing to do with anything that might be called `a` in any other frame.

Some frames are created by the HyperDoc program and these can have pretty strange names, since they are generated automatically. To find out the names of all frames, issue

```
)frame names
```

It will indicate the name of the current frame.

You create a new frame "**quark**" by issuing

```
)frame new quark
```

The history facility can be turned on by issuing either `)set history on` or `)history )on`. If the history facility is on and you are saving history information in a file rather than in the Axiom environment then a history file with filename **quark.axh** will be created as you enter commands. If you wish to go back to what you were doing in the "**initial**" frame, use

```
)frame next
```

or

```
)frame last
```

to cycle through the ring of available frames to get back to "**initial**".

If you want to throw away a frame (say "**quark**"), issue

```
)frame drop quark
```

If you omit the name, the current frame is dropped.

If you do use frames with the history facility on and writing to a file, you may want to delete some of the older history files. These are directories, so you may want to issue a command like `rm -r quark.axh` to the operating system.

You can bring things from another frame by using `)frame import`. For example, to bring the `f` and `g` from the frame "**quark**" to the current frame, issue

```
)frame import quark f g
```

If you want everything from the frame "**quark**", issue

```
)frame import quark
```

You will be asked to verify that you really want everything.

There are two `)set` flags to make it easier to tell where you are.

```
)set message frame on | off
```

will print more messages about frames when it is set on. By default, it is off.

```
)set message prompt frame
```

will give a prompt that looks like

```
initial (1) ->
```

when you start up. In this case, the frame name and step make up the prompt.

Also See: `)history` and `)set` .

1.12   )help

User Level Required: interpreter

Command Syntax:

> `)help`

> `)help` *commandName*

Command Description:

This command displays help information about system commands. If you issue

```
)help
```

then this very text will be shown. You can also give the name or abbreviation of a system command to display information about it. For example,

```
)help clear
```

will display the description of the `)clear` system command.

All this material is available in the Axiom User Guide and in HyperDoc. In HyperDoc, choose the **Commands** item from the **Reference** menu.

1.13   )history

User Level Required: interpreter

Command Syntax:

```
)history )on
```

```
)history )off
```

`)history )write` *historyInputFileName*

`)history )show` [*n*] [both]

`)history )save` *savedHistoryName*

`)history )restore` [*savedHistoryName*]

```
)history )reset
```

`)history )change` *n*

```
)history )memory
```

```
)history )file
```

%

%%(*n*)

```
)set history on | off
```

Command Description:

The *history* facility within Axiom allows you to restore your environment to
that of another session and recall previous computational results. Additional
commands allow you to review previous input lines and to create an **.input** file
of the lines typed to Axiom.

Axiom saves your input and output if the history facility is turned on (which is
the default). This information is saved if either of

```
)set history on
)history )on
```

has been issued. Issuing either

```
)set history off
)history )off
```

will discontinue the recording of information.

Whether the facility is disabled or not, the value of `%` in Axiom always refers
to the result of the last computation. If you have not yet entered anything, `%`
evaluates to an object of type `Variable('%)`. The function `%%` may be used
to refer to other previous results if the history facility is enabled. In that case,
`%%(n)` is the output from step `n` if `n > 0`. If `n < 0`, the step is computed relative
to the current step. Thus `%%(-1)` is also the previous step, `%%(-2)`, is the step

before that, and so on. If an invalid step number is given, Axiom will signal an error.

The *environment* information can either be saved in a file or entirely in memory (the default). Each frame (1.11 on page 1001) has its own history database. When it is kept in a file, some of it may also be kept in memory for efficiency. When the information is saved in a file, the name of the file is of the form **FRAME.axh** where "**FRAME**" is the name of the current frame. The history file is placed in the current working directory (see **??** on page **??**). Note that these history database files are not text files (in fact, they are directories themselves), and so are not in human-readable format.

The options to the)history command are as follows:

)change *n* will set the number of steps that are saved in memory to *n*. This option only has effect when the history data is maintained in a file. If you have issued)history)memory (or not changed the default) there is no need to use)history)change.

)on will start the recording of information. If the workspace is not empty, you will be asked to confirm this request. If you do so, the workspace will be cleared and history data will begin being saved. You can also turn the facility on by issuing)set history on.

)off will stop the recording of information. The)history)show command will not work after issuing this command. Note that this command may be issued to save time, as there is some performance penalty paid for saving the environment data. You can also turn the facility off by issuing)set history off.

)file indicates that history data should be saved in an external file on disk.

)memory indicates that all history data should be kept in memory rather than saved in a file. Note that if you are computing with very large objects it may not be practical to kept this data in memory.

)reset will flush the internal list of the most recent workspace calculations so that the data structures may be garbage collected by the underlying Common Lisp system. Like)history)change, this option only has real effect when history data is being saved in a file.

)restore [*savedHistoryName*] completely clears the environment and restores it to a saved session, if possible. The)save option below allows you to save a session to a file with a given name. If you had issued)history)save jacobi the command)history)restore jacobi would clear the current workspace and load the contents of the named saved session. If no saved session name is specified, the system looks for a file called **last.axh**.

)save *savedHistoryName* is used to save a snapshot of the environment in a file. This file is placed in the current working directory (see **??** on page **??**).

Use `)history )restore` to restore the environment to the state preserved
in the file. This option also creates an input file containing all the lines
of input since you created the workspace frame (for example, by starting
your Axiom session) or last did a `)clear all` or `)clear completely`.

`)show` [*n*] [both] can show previous input lines and output results. `)show` will
display up to twenty of the last input lines (fewer if you haven't typed in
twenty lines). `)show` *n* will display up to *n* of the last input lines. `)show`
`both` will display up to five of the last input lines and output results. `)show`
n `both` will display up to *n* of the last input lines and output results.

`)write` *historyInputFile* creates an **.input** file with the input lines typed
since the start of the session/frame or the last `)clear all` or `)clear`
`completely`. If *historyInputFileName* does not contain a period ("."). in
the filename, **.input** is appended to it. For example, `)history )write`
`chaos` and `)history )write chaos.input` both write the input lines to
a file called **chaos.input** in your current working directory. If you issued
one or more `)undo` commands, `)history )write` eliminates all input lines
backtracked over as a result of `)undo`. You can edit this file and then use
`)read` to have Axiom process the contents.

Also See: `)frame` , `)read` , `)set` , and `)undo` .

1.14   )library

User Level Required: interpreter

Command Syntax:

> `)library` *libName1* [*libName2 ...*]
>
> `)library )dir` *dirName*
>
> `)library )only` *objName1* [*objlib2 ...*]
>
> `)library )noexpose`

Command Description:

This command replaces the `)load` system command that was available in Axiom
releases before version 2.0. The `)library` command makes available to Axiom
the compiled objects in the libraries listed.

For example, if you `)compile dopler.as` in your home directory, issue
`)library dopler` to have Axiom look at the library, determine the category
and domain constructors present, update the internal database with various
properties of the constructors, and arrange for the constructors to be automat-
ically loaded when needed. If the `)noexpose` option has not been given, the
constructors will be exposed (that is, available) in the current frame.

If you compiled a file with the old system compiler, you will have an *NRLIB* present, for example, *DOPLER.NRLIB*, where `DOPLER` is a constructor abbreviation. The command `)library DOPLER` will then do the analysis and database updates as above.

To tell the system about all libraries in a directory, use `)library )dir dirName` where `dirName` is an explicit directory. You may specify "." as the directory, which means the current directory from which you started the system or the one you set via the `)cd` command. The directory name is required.

You may only want to tell the system about particular constructors within a library. In this case, use the `)only` option. The command `)library dopler )only Test1` will only cause the Test1 constructor to be analyzed, autoloaded, etc..

Finally, each constructor in a library are usually automatically exposed when the `)library` command is used. Use the `)noexpose` option if you not want them exposed. At a later time you can use `)set expose add constructor` to expose any hidden constructors.

Note for Axiom beta testers: At various times this command was called `)local` and `)with` before the name `)library` became the official name.

Also See: `)cd` , `)compile` , `)frame` , and `)set` .

1.15   )lisp

User Level Required: development

Command Syntax:

>   `)lisp` [*lispExpression*]

Command Description:

This command is used by Axiom system developers to have single expressions evaluated by the Common Lisp system on which Axiom is built. The *lispExpression* is read by the Common Lisp reader and evaluated. If this expression is not complete (unbalanced parentheses, say), the reader will wait until a complete expression is entered.

Since this command is only useful for evaluating single expressions, the `)fin` command may be used to drop out of Axiom into Common Lisp.

Also See: `)system` , `)boot` , and `)fin` .

1.16   )load

User Level Required: interpreter

Command Description:

This command is obsolete. Use)`library` instead.

1.17   )trace

User Level Required: development

Command Syntax:

This command has the same arguments as options as the)`trace` command.

Command Description:

This command is used by Axiom system developers to trace Common Lisp or BOOT functions. It is not supported for general use.

Also See:)`boot` ,)`lisp` , and)`trace` .

1.18   )pquit

User Level Required: interpreter

Command Syntax:

>)`pquit`

Command Description:

This command is used to terminate Axiom and return to the operating system. Other than by redoing all your computations or by using the)`history`)`restore` command to try to restore your working environment, you cannot return to Axiom in the same state.

)`pquit` differs from the)`quit` in that it always asks for confirmation that you want to terminate Axiom (the "p" is for "protected"). When you enter the)`pquit` command, Axiom responds

> Please enter **y** or **yes** if you really want to leave the interactive
> environment and return to the operating system:

If you respond with `y` or `yes`, you will see the message

> You are now leaving the Axiom interactive environment.
> Issue the command **axiom** to the operating system to start a new session.

and Axiom will terminate and return you to the operating system (or the environment from which you invoked the system). If you responded with something other than `y` or `yes`, then the message

> You have chosen to remain in the Axiom interactive environment.

will be displayed and, indeed, Axiom would still be running.

Also See: `)fin` , `)history` , `)close` , `)quit` , and `)system` .

1.19   )quit

User Level Required: interpreter

Command Syntax:

> `)quit`
>
> `)set quit protected | unprotected`

Command Description:

This command is used to terminate Axiom and return to the operating system. Other than by redoing all your computations or by using the `)history` `)restore` command to try to restore your working environment, you cannot return to Axiom in the same state.

`)quit` differs from the `)pquit` in that it asks for confirmation only if the command

`)set quit protected`

has been issued. Otherwise, `)quit` will make Axiom terminate and return you to the operating system (or the environment from which you invoked the system).

The default setting is `)set quit protected` so that `)quit` and `)pquit` behave in the same way. If you do issue

`)set quit unprotected`

we suggest that you do not (somehow) assign `)quit` to be executed when you press, say, a function key.

Also See: `)fin` , `)history` , `)close` , `)pquit` , and `)system` .

1.20   )read

User Level Required: interpreter

Command Syntax:

> `)read [`*fileName*`]`
>
> `)read [`*fileName*`] [)quiet] [)ifthere]`

Command Description:

This command is used to read **.input** files into Axiom. The command

```
)read matrix.input
```

will read the contents of the file **matrix.input** into Axiom. The ".input" file
extension is optional. See 4.1 on page 183 for more information about **.input**
files.

This command remembers the previous file you edited, read or compiled. If you
do not specify a file name, the previous file will be read.

The)**ifthere** option checks to see whether the **.input** file exists. If it does
not, the)**read** command does nothing. If you do not use this option and the
file does not exist, you are asked to give the name of an existing **.input** file.

The)**quiet** option suppresses output while the file is being read.

Also See:)compile ,)edit , and)history .

1.21   )set

User Level Required: interpreter

Command Syntax:

>)set

>)set *label1* [*... labelN*]

>)set *label1* [*... labelN*] *newValue*

Command Description:

The)**set** command is used to view or set system variables that control what
messages are displayed, the type of output desired, the status of the history
facility, the way Axiom user functions are cached, and so on. Since this collection
is very large, we will not discuss them here. Rather, we will show how the facility
is used. We urge you to explore the)**set** options to familiarize yourself with
how you can modify your Axiom working environment. There is a HyperDoc
version of this same facility available from the main HyperDoc menu.

The)**set** command is command-driven with a menu display.   It is tree-
structured. To see all top-level nodes, issue)**set** by itself.

```
)set
```

Variables with values have them displayed near the right margin. Subtrees of
selections have "..." displayed in the value field. For example, there are many
kinds of messages, so issue)**set message** to see the choices.

```
)set message
```

The current setting for the variable that displays whether computation times are displayed is visible in the menu displayed by the last command. To see more information, issue

```
)set message time
```

This shows that time printing is on now. To turn it off, issue

```
)set message time off
```

As noted above, not all settings have so many qualifiers. For example, to change the `)quit` command to being unprotected (that is, you will not be prompted for verification), you need only issue

```
)set quit unprotected
```

Also See: `)quit` .

1.22   )show

User Level Required: interpreter

Command Syntax:

> `)show` *nameOrAbbrev*
>
> `)show` *nameOrAbbrev* `)operations`
>
> `)show` *nameOrAbbrev* `)attributes`

Command Description: This command displays information about Axiom domain, package and category *constructors*. If no options are given, the `)operations` option is assumed. For example,

```
)show POLY
)show POLY )operations
)show Polynomial
)show Polynomial )operations
```

each display basic information about the `Polynomial` domain constructor and then provide a listing of operations. Since `Polynomial` requires a `Ring` (for example, `Integer`) as argument, the above commands all refer to a unspecified ring R. In the list of operations, `$` means `Polynomial(R)`.

The basic information displayed includes the *signature* of the constructor (the name and arguments), the constructor *abbreviation*, the *exposure status* of the constructor, and the name of the *library source file* for the constructor.

If operation information about a specific domain is wanted, the full or abbreviated domain name may be used. For example,

```
)show POLY INT
)show POLY INT )operations
)show Polynomial Integer
)show Polynomial Integer )operations
```

are among the combinations that will display the operations exported by the domain `Polynomial(Integer)` (as opposed to the general *domain constructor* `Polynomial`). Attributes may be listed by using the `)attributes` option.

Also See: `)display` , `)set` , and `)what` .

1.23   )spool

User Level Required: interpreter

Command Syntax:

> `)spool` [*fileName*]

> `)spool`

Command Description:

This command is used to save *(spool)* all Axiom input and output into a file, called a *spool file.* You can only have one spool file active at a time. To start spool, issue this command with a filename. For example,

```
)spool integrate.out
```

To stop spooling, issue `)spool` with no filename.

If the filename is qualified with a directory, then the output will be placed in that directory. If no directory information is given, the spool file will be placed in the *current directory.* The current directory is the directory from which you started Axiom or is the directory you specified using the `)cd` command.

Also See: `)cd` .

1.24   )synonym

User Level Required: interpreter

Command Syntax:

> `)synonym`

> `)synonym` *synonym fullCommand*

> `)what synonyms`

Command Description:

This command is used to create short synonyms for system command expressions. For example, the following synonyms might simplify commands you often use.

```
)synonym save        history )save
)synonym restore     history )restore
)synonym mail        system mail
)synonym ls          system ls
)synonym fortran     set output fortran
```

Once defined, synonyms can be used in place of the longer command expressions. Thus

```
)fortran on
```

is the same as the longer

```
)set fortran output on
```

To list all defined synonyms, issue either of

```
)synonyms
)what synonyms
```

To list, say, all synonyms that contain the substring "**ap**", issue

```
)what synonyms ap
```

Also See:)set and)what .

1.25   )system

User Level Required: interpreter

Command Syntax:

>)system *cmdExpression*

Command Description:

This command may be used to issue commands to the operating system while remaining in Axiom. The *cmdExpression* is passed to the operating system for execution.

To get an operating system shell, issue, for example,)system sh. When you enter the key combination, | **Ctrl** |–| **D** | (pressing and holding the | **Ctrl** | key and

then pressing the $\boxed{\mathbf{D}}$ key) the shell will terminate and you will return to Axiom. We do not recommend this way of creating a shell because Common Lisp may field some interrupts instead of the shell. If possible, use a shell running in another window.

If you execute programs that misbehave you may not be able to return to Axiom. If this happens, you may have no other choice than to restart Axiom and restore the environment via `)history )restore`, if possible.

Also See: `)boot` , `)fin` , `)lisp` , `)pquit` , and `)quit` .

1.26   )trace

User Level Required: interpreter

Command Syntax:

> `)trace`
>
> `)trace )off`
>
> `)trace` *function* [*options*]
>
> `)trace` *constructor* [*options*]
>
> `)trace` *domainOrPackage* [*options*]

where options can be one or more of

> `)after` *S-expression*
>
> `)before` *S-expression*
>
> `)break after`
>
> `)break before`
>
> `)cond` *S-expression*
>
> `)count`
>
> `)count` *n*
>
> `)depth` *n*
>
> `)local` *op1* [*... opN*]
>
> `)nonquietly`
>
> `)nt`
>
> `)off`

)only *listOfDataToDisplay*

)ops

)ops *op1* [*... opN*]

)restore

)stats

)stats reset

)timer

)varbreak

)varbreak *var1* [*... varN*]

)vars

)vars *var1* [*... varN*]

)within *executingFunction*

Command Description:

This command is used to trace the execution of functions that make up the
Axiom system, functions defined by users, and functions from the system library.
Almost all options are available for each type of function but exceptions will be
noted below.

To list all functions, constructors, domains and packages that are traced, simply
issue

)trace

To untrace everything that is traced, issue

)trace)off

When a function is traced, the default system action is to display the arguments
to the function and the return value when the function is exited. Note that if a
function is left via an action such as a `THROW`, no return value will be displayed.
Also, optimization of tail recursion may decrease the number of times a function
is actually invoked and so may cause less trace information to be displayed.
Other information can be displayed or collected when a function is traced and
this is controlled by the various options. Most options will be of interest only to
Axiom system developers. If a domain or package is traced, the default action
is to trace all functions exported.

Individual interpreter, lisp or boot functions can be traced by listing their names
after)trace. Any options that are present must follow the functions to be
traced.

```
)trace f
```

traces the function `f`. To untrace `f`, issue

```
)trace f )off
```

Note that if a function name contains a special character, it will be necessary to escape the character with an underscore

```
)trace _/D_,1
```

To trace all domains or packages that are or will be created from a particular constructor, give the constructor name or abbreviation after `)trace`.

```
)trace MATRIX
)trace List Integer
```

The first command traces all domains currently instantiated with `Matrix`. If additional domains are instantiated with this constructor (for example, if you have used `Matrix(Integer)` and `Matrix(Float)`), they will be automatically traced. The second command traces `List(Integer)`. It is possible to trace individual functions in a domain or package. See the `)ops` option below.

The following are the general options for the `)trace` command.

`)break after` causes a Common Lisp break loop to be entered after exiting the traced function.

`)break before` causes a Common Lisp break loop to be entered before entering the traced function.

`)break` is the same as `)break before`.

`)count` causes the system to keep a count of the number of times the traced function is entered. The total can be displayed with `)trace )stats` and cleared with `)trace )stats reset`.

`)count` n causes information about the traced function to be displayed for the first n executions. After the *n-th execution, the function is untraced.*

`)depth` n *causes trace information to be shown for only n levels of recursion of the traced function. The command*

```
)trace fib )depth 10
```

will cause the display of only 10 levels of trace information for the recursive execution of a user function **fib**.

)math *causes the function arguments and return value to be displayed in the Axiom monospace two-dimensional math format.*

)nonquietly *causes the display of additional messages when a function is traced.*

)nt *This suppresses all normal trace information. This option is useful if the*)count *or*)timer *options are used and you are interested in the statistics but not the function calling information.*

)off *causes untracing of all or specific functions. Without an argument, all functions, constructors, domains and packages are untraced. Otherwise, the given functions and other objects are untraced. To immediately retrace the untraced functions, issue*)trace)restore*.*

)only *listOfDataToDisplay causes only specific trace information to be shown. The items are listed by using the following abbreviations:*

   a *display all arguments*

   v *display return value*

   1 *display first argument*

   2 *display second argument*

   15 *display the 15th argument, and so on*

)restore causes the last untraced functions to be retraced. If additional options are present, they are added to those previously in effect.

)stats causes the display of statistics collected by the use of the)count and)timer options.

)stats reset resets to 0 the statistics collected by the use of the)count and)timer options.

)timer causes the system to keep a count of execution times for the traced function. The total can be displayed with)trace)stats and cleared with)trace)stats reset.

)varbreak *var1* [*... varN*] causes a Common Lisp break loop to be entered after the assignment to any of the listed variables in the traced function.

)vars causes the display of the value of any variable after it is assigned in the traced function. Note that library code must have been compiled (see **??** on page **??**) using the)vartrace option in order to support this option.

)vars *var1* [*... varN*] causes the display of the value of any of the specified variables after they are assigned in the traced function. Note that library code must have been compiled (see **??** on page **??**) using the)vartrace option in order to support this option.

)within *executingFunction* causes the display of trace information only if the
traced function is called when the given *executingFunction* is running.

The following are the options for tracing constructors, domains and packages.

)local [*op1* [*... opN*]] causes local functions of the constructor to be traced.
Note that to untrace an individual local function, you must use the fully
qualified internal name, using the escape character _ before the semicolon.

```
)trace FRAC )local
)trace FRAC_;cancelGcd )off
```

)ops *op1* [*... opN*] By default, all operations from a domain or package are
traced when the domain or package is traced. This option allows you to
specify that only particular operations should be traced. The command

```
)trace Integer )ops min max _+ _-
```

traces four operations from the domain `Integer`. Since + and - are special
characters, it is necessary to escape them with an underscore.

Also See:)boot ,)lisp , and)ltrace .

1.27   )undo

User Level Required: interpreter

Command Syntax:

>)undo

>)undo *integer*

>)undo *integer* [*option*]

>)undo)redo

where *option* is one of

>)after

>)before

Command Description:

This command is used to restore the state of the user environment to an earlier
point in the interactive session. The argument of an)undo is an integer which
must designate some step number in the interactive session.

```
)undo n
)undo n )after
```

These commands return the state of the interactive environment to that immediately after step `n`. If `n` is a positive number, then `n` refers to step nummber `n`. If `n` is a negative number, it refers to the `n`-th previous command (that is, undoes the effects of the last $-n$ commands).

A `)clear all` resets the `)undo` facility. Otherwise, an `)undo` undoes the effect of `)clear` with options `properties`, `value`, and `mode`, and that of a previous `undo`. If any such system commands are given between steps n and $n+1$ $(n > 0)$, their effect is undone for `)undo m` for any $0 < m \leq n..$

The command `)undo` is equivalent to `)undo -1` (it undoes the effect of the previous user expression). The command `)undo 0` undoes any of the above system commands issued since the last user expression.

```
)undo n )before
```

This command returns the state of the interactive environment to that immediately before step `n`. Any `)undo` or `)clear` system commands given before step `n` will not be undone.

```
)undo )redo
```

This command reads the file `redo.input.` created by the last `)undo` command. This file consists of all user input lines, excluding those backtracked over due to a previous `)undo`.

Also See: `)history` . The command `)history )write` will eliminate the "undone" command lines of your program.

1.28   )what

User Level Required: interpreter

Command Syntax:

> `)what categories` *pattern1* [*pattern2 ...*]
>
> `)what commands`   *pattern1* [*pattern2 ...*]
>
> `)what domains`   *pattern1* [*pattern2 ...*]
>
> `)what operations` *pattern1* [*pattern2 ...*]
>
> `)what packages`   *pattern1* [*pattern2 ...*]
>
> `)what synonym`   *pattern1* [*pattern2 ...*]

)what things   *pattern1* [*pattern2 ...*]

)apropos   *pattern1* [*pattern2 ...*]

Command Description:

This command is used to display lists of things in the system. The patterns are all strings and, if present, restrict the contents of the lists. Only those items that contain one or more of the strings as substrings are displayed. For example,

```
)what synonym
```

displays all command synonyms,

```
)what synonym ver
```

displays all command synonyms containing the substring "`ver`",

```
)what synonym ver pr
```

displays all command synonyms containing the substring "`ver`" or the substring "`pr`". Output similar to the following will be displayed

```
---------------- System Command Synonyms -----------------

user-defined synonyms satisfying patterns:
      ver pr

  )apr ........................... )what things
  )apropos ....................... )what things
  )prompt ........................ )set message prompt
  )version ....................... )lisp *yearweek*
```

Several other things can be listed with the)`what` command:

`categories` displays a list of category constructors.

`commands` displays a list of system commands available at your user-level. Your user-level is set via the)`set userlevel` command. To get a description of a particular command, such as ")`what`", issue)`help what`.

`domains` displays a list of domain constructors.

`operations` displays a list of operations in the system library. It is recommended that you qualify this command with one or more patterns, as there are thousands of operations available. For example, say you are looking for functions that involve computation of eigenvalues. To find their names, try)`what operations eig`. A rather large list of operations is loaded into the workspace when this command is first issued. This list will be deleted when you clear the workspace via)`clear all` or)`clear completely`. It will be re-created if it is needed again.

`packages` displays a list of package constructors.

`synonym` lists system command synonyms.

`things` displays all of the above types for items containing the pattern strings as substrings. The command synonym `)apropos` is equivalent to `)what things`.

Also See: `)display` , `)set` , and `)show` .

Chapter 2

Categories

This is a listing of all categories in the Axiom library at the time this book was produced. Use the Browse facility (described in Chapter 14 on page 943) to get more information about these constructors.

This sample entry will help you read the following table:

CategoryNameCategoryAbbreviation:$\mathsf{Category}_1 \ldots \mathsf{Category}_N$ *with* $\mathrm{op}_1 \ldots \mathrm{op}_M$

where

CategoryName	is the full category name, e.g., CommutativeRing.
CategoryAbbreviation	is the category abbreviation, e.g., COMRING.
$\mathsf{Category}_i$	is a category to which the category belongs.
op_j	is an operation exported by the category.

Categories

Appendix A

constructorListing

ABELGRP AbelianGroup CancellationAbelianMonoid * – AMR AbelianMonoidRing Algebra BiModule CharacteristicNonZero CharacteristicZero CommutativeRing IntegralDomain Ring / coefficient degree leadingCoefficient leadingMonomial map monomial monomial? reductum ABELMON AbelianMonoid AbelianSemiGroup * Zero zero? ABELSG AbelianSemiGroup SetCategory * + AGG Aggregate Object # copy empty empty? eq? less? more? size? ACF AlgebraicallyClosedField Field RadicalCategory rootOf rootsOf zeroOf zerosOf ACFS AlgebraicallyClosedFunctionSpace AlgebraicallyClosedField FunctionSpace rootOf rootsOf zeroOf zerosOf ALGEBRA Algebra Module Ring coerce AHYP ArcHyperbolicFunctionCategory acosh acoth acsch asech asinh atanh ATRIG ArcTrigonometricFunctionCategory acos acot acsc asec asin atan ALAGG AssociationListAggregate ListAggregate TableAggregate assoc ATTREG AttributeRegistry BGAGG BagAggregate HomogeneousAggregate bag extract! insert! inspect BMODULE BiModule LeftModule RightModule BRAGG BinaryRecursiveAggregate RecursiveAggregate elt left right setelt setleft! setright! BTCAT BinaryTreeCategory BinaryRecursiveAggregate node BTAGG BitAggregate OneDimensionalArrayAggregate OrderedSet ^ and nand nor not or xor CACHSET CachableSet OrderedSet position setPosition CABMON CancellationAbelianMonoid AbelianMonoid – CHARNZ CharacteristicNonZero Ring charthRoot CHARZ CharacteristicZero Ring KOERCE CoercibleTo coerce CLAGG Collection ConvertibleTo HomogeneousAggregate construct find reduce remove removeDuplicates select CFCAT CombinatorialFunctionCategory binomial factorial permutation COMBOPC CombinatorialOpsCategory CombinatorialFunctionCategory factorials product summation COMRING CommutativeRing BiModule Ring COMPCAT ComplexCategory CharacteristicNonZero CharacteristicZero CommutativeRing

ConvertibleTo DifferentialExtension EuclideanDomain Field FullyE-valableOver FullyLinearlyExplicitRingOver FullyRetractableTo IntegralDomain MonogenicAlgebra OrderedSet PolynomialFactorizationExplicit RadicalCategory TranscendentalFunctionCategory abs argument complex conjugate exquo imag imaginary norm polarCoordinates rational rational? rationalIfCan real KONVERT ConvertibleTo convert DQAGG DequeueAggregate QueueAggregate StackAggregate bottom! dequeue extractBottom! extractTop! height insertBottom! insertTop! reverse! top! DIOPS DictionaryOperations BagAggregate Collection dictionary remove! select! DIAGG Dictionary DictionaryOperations DIFEXT DifferentialExtension DifferentialRing PartialDifferentialRing Ring D differentiate DPOLCAT DifferentialPolynomialCategory DifferentialExtension Evalable InnerEvalable PolynomialCategory RetractableTo degree differentialVariables initial isobaric? leader makeVariable order separant weight weights DIFRING DifferentialRing Ring D differentiate DVARCAT DifferentialVariableCategory OrderedSet RetractableTo D coerce differentiate makeVariable order variable weight DIRPCAT DirectProductCategory AbelianSemiGroup Algebra BiModule CancellationAbelianMonoid CoercibleTo CommutativeRing DifferentialExtension Finite FullyLinearlyExplicitRingOver FullyRetractableTo IndexedAggregate OrderedAbelianMonoidSup OrderedRing VectorSpace * directProduct dot unitVector DIVRING DivisionRing Algebra EntireRing ** inv DLAGG DoublyLinkedAggregate RecursiveAggregate concat! head last next previous setnext! setprevious! tail ELEMFUN ElementaryFunctionCategory ** exp log ELTAGG EltableAggregate Eltable elt qelt qsetelt! setelt ELTAB Eltable elt ENTIRER EntireRing BiModule Ring EUCDOM EuclideanDomain PrincipalIdealDomain divide euclideanSize extendedEuclidean multiEuclidean quo rem sizeLess? EVALAB Evalable eval ES ExpressionSpace Evalable InnerEvalable OrderedSet RetractableTo belong? box definingPolynomial distribute elt eval freeOf? height is? kernel kernels mainKernel map minPoly operator operators paren subst tower ELAGG ExtensibleLinearAggregate LinearAggregate concat! delete! insert! merge! remove! removeDuplicates! select! XF ExtensionField CharacteristicZero Field FieldOfPrimeCharacteristic RetractableTo VectorSpace Frobenius algebraic? degree extensionDegree inGroundField? transcendenceDegree transcendent? FPC FieldOfPrimeCharacteristic CharacteristicNonZero Field discreteLog order primeFrobenius FIELD Field DivisionRing EuclideanDomain UniqueFactorizationDomain / FILECAT FileCategory SetCategory close! iomode name open read! reopen! write! FNCAT FileNameCategory SetCategory coerce directory exists? extension filename name new readable? writable? FAMR FiniteAbelianMonoidRing AbelianMonoidRing FullyRetractableTo coefficients content exquo ground ground? mapExponents minimumDegree numberOfMono-

mials primitivePart FAXF FiniteAlgebraicExtensionField Extension-
Field FiniteFieldCategory RetractableTo basis coordinates creat-
eNormalElement definingPolynomial degree extensionDegree gen-
erator minimalPolynomial norm normal? normalElement repre-
sents trace FFIELDC FiniteFieldCategory FieldOfPrimeCharacteris-
tic Finite StepThrough charthRoot conditionP createPrimitiveEle-
ment discreteLog factorsOfCyclicGroupSize order primitive? primi-
tiveElement representationType tableForDiscreteLogarithm FLAGG
FiniteLinearAggregate LinearAggregate OrderedSet copyInto! merge
position reverse reverse! sort sort! sorted? FINRALG FiniteR-
ankAlgebra Algebra CharacteristicNonZero CharacteristicZero char-
acteristicPolynomial coordinates discriminant minimalPolynomial
norm rank regularRepresentation represents trace traceMatrix FI-
NAALG FiniteRankNonAssociativeAlgebra NonAssociativeAlgebra
JacobiIdentity? JordanAlgebra? alternative? antiAssociative? an-
tiCommutative? associative? associatorDependence commutative?
conditionsForIdempotents coordinates flexible? jordanAdmissible?
leftAlternative? leftCharacteristicPolynomial leftDiscriminant left-
MinimalPolynomial leftNorm leftRecip leftRegularRepresentation
leftTrace leftTraceMatrix leftUnit leftUnits lieAdmissible? lieAlge-
bra? noncommutativeJordanAlgebra? powerAssociative? rank recip
represents rightAlternative? rightCharacteristicPolynomial rightDis-
criminant rightMinimalPolynomial rightNorm rightRecip rightRegu-
larRepresentation rightTrace rightTraceMatrix rightUnit rightUnits
someBasis structuralConstants unit FSAGG FiniteSetAggregate Dic-
tionary Finite SetAggregate cardinality complement max min uni-
verse FINITE Finite SetCategory index lookup random size FPS
FloatingPointSystem RealNumberSystem base bits decreasePrecision
digits exponent float increasePrecision mantissa max order preci-
sion FRAMALG FramedAlgebra FiniteRankAlgebra basis convert
coordinates discriminant regularRepresentation represents traceMa-
trix FRNAALG FramedNonAssociativeAlgebra FiniteRankNonAsso-
ciativeAlgebra apply basis conditionsForIdempotents convert coordi-
nates elt leftDiscriminant leftRankPolynomial leftRegularRepresen-
tation leftTraceMatrix represents rightDiscriminant rightRankPoly-
nomial rightRegularRepresentation rightTraceMatrix structuralCon-
stants FAMONC FreeAbelianMonoidCategory CancellationAbelian-
Monoid RetractableTo * + coefficient highCommonTerms map-
Coef mapGen nthCoef nthFactor size terms FEVALAB FullyEval-
ableOver Eltable Evalable InnerEvalable map FLINEXP FullyLin-
earlyExplicitRingOver LinearlyExplicitRingOver FPATMAB Fully-
PatternMatchable Object PatternMatchable FRETRCT FullyRe-
tractableTo RetractableTo FFCAT FunctionFieldCategory Mono-
genicAlgebra D absolutelyIrreducible? branchPoint? branchPointAt-
Infinity? complementaryBasis differentiate elt genus integral? in-
tegralAtInfinity? integralBasis integralBasisAtInfinity integralCo-

ordinates integralDerivationMatrix integralMatrix integralMatrixAt-Infinity integralRepresents inverseIntegralMatrix inverseIntegralMatrixAtInfinity nonSingularModel normalizeAtInfinity numberOfComponents primitivePart ramified? ramifiedAtInfinity? rationalPoint? rationalPoints reduceBasisAtInfinity represents singular? singularAtInfinity? yCoordinates FS FunctionSpace AbelianGroup AbelianMonoid Algebra CharacteristicNonZero CharacteristicZero ConvertibleTo ExpressionSpace Field FullyLinearlyExplicitRingOver FullyPatternMatchable FullyRetractableTo Group Monoid PartialDifferentialRing Patternable RetractableTo Ring ** / applyQuote coerce convert denom denominator eval ground ground? isExpt isMult isPlus isPower isTimes numer numerator univariate variables GCDDOM GcdDomain IntegralDomain gcd lcm GRALG GradedAlgebra GradedModule One product GRMOD GradedModule RetractableTo SetCategory * + – Zero degree GROUP Group Monoid ** / commutator conjugate inv HOAGG HomogeneousAggregate Aggregate SetCategory any? count every? map map! member? members parts HYPCAT HyperbolicFunctionCategory cosh coth csch sech sinh tanh IXAGG IndexedAggregate EltableAggregate HomogeneousAggregate entries entry? fill! first index? indices maxIndex minIndex swap! IDPC IndexedDirectProductCategory SetCategory leadingCoefficient leadingSupport map monomial reductum IEVALAB InnerEvalable eval INS IntegerNumberSystem CharacteristicZero CombinatorialFunctionCategory ConvertibleTo DifferentialRing EuclideanDomain LinearlyExplicitRingOver OrderedRing PatternMatchable RealConstant RetractableTo StepThrough UniqueFactorizationDomain addmod base bit? copy dec even? hash inc invmod length mask mulmod odd? positiveRemainder powmod random rational rational? rationalIfCan shift submod symmetricRemainder INTDOM IntegralDomain Algebra CommutativeRing EntireRing associates? exquo unit? unitCanonical unitNormal KDAGG KeyedDictionary Dictionary key? keys remove! search LZSTAGG LazyStreamAggregate StreamAggregate complete explicitEntries? explicitlyEmpty? extend frst lazy? lazyEvaluate numberOfComputedEntries remove rst select LALG LeftAlgebra LeftModule Ring coerce LMODULE LeftModule AbelianGroup * LNAGG LinearAggregate Collection IndexedAggregate concat delete elt insert map new setelt LINEXP LinearlyExplicitRingOver Ring reducedSystem LFCAT LiouvillianFunctionCategory PrimitiveFunctionCategory TranscendentalFunctionCategory Ci Ei Si dilog erf li LSAGG ListAggregate ExtensibleLinearAggregate FiniteLinearAggregate StreamAggregate list MATCAT MatrixCategory TwoDimensionalArrayCategory * ** + – / antisymmetric? coerce determinant diagonal? diagonalMatrix elt exquo horizConcat inverse listOfLists matrix minordet nullSpace nullity rank rowEchelon scalarMatrix setelt setsubMatrix! square? squareTop subMatrix swapColumns! swapRows!

symmetric? transpose vertConcat zero MODULE Module BiModule MONADWU MonadWithUnit Monad ∗∗ One leftPower leftRecip one? recip rightPower rightRecip MONAD Monad SetCategory ∗ ∗∗ leftPower rightPower MONOGEN MonogenicAlgebra CommutativeRing ConvertibleTo DifferentialExtension Field Finite FiniteFieldCategory FramedAlgebra FullyLinearlyExplicitRingOver FullyRetractableTo convert definingPolynomial derivationCoordinates generator lift reduce MLO MonogenicLinearOperator Algebra BiModule Ring coefficient degree leadingCoefficient minimumDegree monomial reductum MONOID Monoid SemiGroup ∗∗ One one? recip MDAGG MultiDictionary DictionaryOperations duplicates insert! removeDuplicates! MSAGG MultiSetAggregate MultiDictionary SetAggregate MTSCAT MultivariateTaylorSeriesCategory Evalable InnerEvalable PartialDifferentialRing PowerSeriesCategory RadicalCategory TranscendentalFunctionCategory coefficient extend integrate monomial order polynomial NAALG NonAssociativeAlgebra Module NonAssociativeRng plenaryPower NASRING NonAssociativeRing MonadWithUnit NonAssociativeRng characteristic coerce NARNG NonAssociativeRng AbelianGroup Monad antiCommutator associator commutator OBJECT Object OC OctonionCategory Algebra CharacteristicNonZero CharacteristicZero ConvertibleTo Finite FullyEvalableOver FullyRetractableTo OrderedSet abs conjugate imagE imagI imagJ imagK imagi imagj imagk inv norm octon rational rational? rationalIfCan real A1AGG OneDimensionalArrayAggregate FiniteLinearAggregate OAGROUP OrderedAbelianGroup AbelianGroup OrderedCancellationAbelianMonoid OAMONS OrderedAbelianMonoidSup OrderedCancellationAbelianMonoid sup OAMON OrderedAbelianMonoid AbelianMonoid OrderedAbelianSemiGroup OASGP OrderedAbelianSemiGroup AbelianMonoid OrderedSet OCAMON OrderedCancellationAbelianMonoid CancellationAbelianMonoid OrderedAbelianMonoid ORDFIN OrderedFinite Finite OrderedSet ORDMON OrderedMonoid Monoid OrderedSet OMAGG OrderedMultiSetAggregate MultiSetAggregate PriorityQueueAggregate min ORDRING OrderedRing OrderedAbelianGroup OrderedMonoid Ring abs negative? positive? sign ORDSET OrderedSet SetCategory < max min PADICCT PAdicIntegerCategory CharacteristicZero EuclideanDomain approximate complete digits extend moduloP modulus order quotientByP sqrt PDRING PartialDifferentialRing Ring D differentiate PTRANFN PartialTranscendentalFunctions acosIfCan acoshIfCan acotIfCan acothIfCan acscIfCan acschIfCan asecIfCan asechIfCan asinIfCan asinhIfCan atanIfCan atanhIfCan cosIfCan coshIfCan cotIfCan cothIfCan cscIfCan cschIfCan expIfCan logIfCan nthRootIfCan secIfCan sechIfCan sinIfCan sinhIfCan tanIfCan tanhIfCan PATAB Patternable ConvertibleTo Object PATMAB PatternMatchable SetCategory patternMatch PERMCAT PermutationCate-

gory Group OrderedSet < cycle cycles elt eval orbit PPCURVE Plot-
tablePlaneCurveCategory CoercibleTo listBranches xRange yRange
PSCURVE PlottableSpaceCurveCategory CoercibleTo listBranches
xRange yRange zRange PTCAT PointCategory VectorCategory con-
vert cross dimension extend length point POLYCAT Polynomi-
alCategory ConvertibleTo Evalable FiniteAbelianMonoidRing Ful-
lyLinearlyExplicitRingOver GcdDomain InnerEvalable OrderedSet
PartialDifferentialRing PatternMatchable PolynomialFactorization-
Explicit RetractableTo coefficient content degree discriminant isExpt
isPlus isTimes mainVariable minimumDegree monicDivide monomial
monomials multivariate primitiveMonomials primitivePart resultant
squareFree squareFreePart totalDegree univariate variables PFECAT
PolynomialFactorizationExplicit UniqueFactorizationDomain charth-
Root conditionP factorPolynomial factorSquareFreePolynomial gcd-
Polynomial solveLinearPolynomialEquation squareFreePolynomial
PSCAT PowerSeriesCategory AbelianMonoidRing complete mono-
mial pole? variables PRIMCAT PrimitiveFunctionCategory in-
tegral PID PrincipalIdealDomain GcdDomain expressIdealMember
principalIdeal PRQAGG PriorityQueueAggregate BagAggregate max
merge merge! QUATCAT QuaternionCategory Algebra Charac-
teristicNonZero CharacteristicZero ConvertibleTo DifferentialExten-
sion DivisionRing EntireRing FullyEvalableOver FullyLinearlyEx-
plicitRingOver FullyRetractableTo OrderedSet abs conjugate imagI
imagJ imagK norm quatern rational rational? rationalIfCan real
QUAGG QueueAggregate BagAggregate back dequeue! enqueue!
front length rotate! QFCAT QuotientFieldCategory Algebra Char-
acteristicNonZero CharacteristicZero ConvertibleTo DifferentialEx-
tension Field FullyEvalableOver FullyLinearlyExplicitRingOver Ful-
lyPatternMatchable OrderedRing OrderedSet Patternable Polynomi-
alFactorizationExplicit RealConstant RetractableTo StepThrough /
ceiling denom denominator floor fractionPart numer numerator ran-
dom wholePart RADCAT RadicalCategory ** nthRoot sqrt REAL
RealConstant ConvertibleTo RNS RealNumberSystem Characteris-
ticZero ConvertibleTo Field OrderedRing PatternMatchable Radical-
Category RealConstant RetractableTo abs ceiling floor fractionPart
norm round truncate wholePart RMATCAT RectangularMatrixCat-
egory BiModule HomogeneousAggregate Module / antisymmetric?
column diagonal? elt exquo listOfLists map matrix maxColIndex
maxRowIndex minColIndex minRowIndex ncols nrows nullSpace nul-
lity qelt rank row rowEchelon square? symmetric? RCAGG Re-
cursiveAggregate HomogeneousAggregate children cyclic? elt leaf?
leaves node? nodes setchildren! setelt setvalue! value RETRACT
RetractableTo coerce retract retractIfCan RMODULE RightMod-
ule AbelianGroup * RING Ring LeftModule Monoid Rng charac-
teristic coerce RNG Rng AbelianGroup SemiGroup SEGCAT Seg-
mentCategory SetCategory BY SEGMENT convert hi high incr lo

low segment SEGXCAT SegmentExpansionCategory SegmentCategory expand map SGROUP SemiGroup SetCategory * ** SETAGG SetAggregate Collection SetCategory < brace difference intersect subset? symmetricDifference union SETCAT SetCategory CoercibleTo Object = SEXCAT SExpressionCategory SetCategory # atom? car cdr convert destruct elt eq expr float float? integer integer? list? null? pair? string string? symbol symbol? uequal SPFCAT SpecialFunctionCategory Beta Gamma abs airyAi airyBi besselI besselJ besselK besselY digamma polygamma SMATCAT SquareMatrixCategory Algebra BiModule DifferentialExtension FullyLinearlyExplicitRingOver FullyRetractableTo Module RectangularMatrixCategory * ** determinant diagonal diagonalMatrix diagonalProduct inverse minordet scalarMatrix trace SKAGG StackAggregate BagAggregate depth pop! push! top STEP StepThrough SetCategory init nextItem STAGG StreamAggregate LinearAggregate UnaryRecursiveAggregate explicitlyFinite? possiblyInfinite? SRAGG StringAggregate OneDimensionalArrayAggregate coerce elt leftTrim lowerCase lowerCase! match match? position prefix? replace rightTrim split substring? suffix? trim upperCase upperCase! STRICAT StringCategory StringAggregate string TBAGG TableAggregate IndexedAggregate KeyedDictionary map setelt table SPACEC ThreeSpaceCategory SetCategory check closedCurve closedCurve? coerce components composite composites copy create3Space curve curve? enterPointData lllip lllp llprop lp lprop merge mesh mesh? modifyPointData numberOfComponents numberOfComposites objects point point? polygon polygon? subspace TRANFUN TranscendentalFunctionCategory ArcHyperbolicFunctionCategory ArcTrigonometricFunctionCategory ElementaryFunctionCategory HyperbolicFunctionCategory TrigonometricFunctionCategory pi TRIGCAT TrigonometricFunctionCategory cos cot csc sec sin tan ARR2CAT TwoDimensionalArrayCategory HomogeneousAggregate column elt fill! map map! maxColIndex maxRowIndex minColIndex minRowIndex ncols new nrows parts qelt qsetelt! row setColumn! setRow! setelt URAGG UnaryRecursiveAggregate RecursiveAggregate concat concat! cycleEntry cycleLength cycleSplit! cycleTail elt first last rest second setelt setfirst! setlast! setrest! split! tail third UFD UniqueFactorizationDomain GcdDomain factor prime? squareFree squareFreePart ULSCAT UnivariateLaurentSeriesCategory Field RadicalCategory TranscendentalFunctionCategory UnivariatePowerSeriesCategory integrate multiplyCoefficients rationalFunction ULSCCAT UnivariateLaurentSeriesConstructorCategory QuotientFieldCategory RetractableTo UnivariateLaurentSeriesCategory coerce degree laurent removeZeroes taylor taylorIfCan taylorRep UPOLYC UnivariatePolynomialCategory DifferentialExtension DifferentialRing Eltable EuclideanDomain PolynomialCategory StepThrough D composite differentiate discriminant

divideExponents elt integrate makeSUP monicDivide multiplyExponents order pseudoDivide pseudoQuotient pseudoRemainder resultant separate subResultantGcd unmakeSUP vectorise UPSCAT UnivariatePowerSeriesCategory DifferentialRing Eltable PowerSeriesCategory approximate center elt eval extend multiplyExponents order series terms truncate variable UPXSCAT UnivariatePuiseuxSeriesCategory Field RadicalCategory TranscendentalFunctionCategory UnivariatePowerSeriesCategory integrate multiplyExponents UPXSCCA UnivariatePuiseuxSeriesConstructorCategory RetractableTo UnivariatePuiseuxSeriesCategory coerce degree laurent laurentIfCan laurentRep puiseux rationalPower UTSCAT UnivariateTaylorSeriesCategory RadicalCategory TranscendentalFunctionCategory UnivariatePowerSeriesCategory ∗∗ coefficients integrate multiplyCoefficients polynomial quoByVar series VECTCAT VectorCategory OneDimensionalArrayAggregate ∗ + − dot zero VSPACE VectorSpace Module / dimension

Appendix C

Domains

This is a listing of all domains in the Axiom library at the time this book was produced. Use the Browse facility (described in Chapter 14 on page 943) to get more information about these constructors.

This sample entry will help you read the following table:

DomainNameDomainAbbreviation:$\mathsf{Category}_1\ldots\mathsf{Category}_N\,with\,\mathrm{op}_1\ldots\mathrm{op}_M$

where

DomainName	is the full domain name, e.g., Integer.
DomainAbbreviation	is the domain abbreviation, e.g., INT.
$\mathsf{Category}_i$	is a category to which the domain belongs.
op_j	is an operation exported by the domain.

Domains

ALGSC AlgebraGivenByStructuralConstants FramedNonAssociativeAlgebra LeftModule 0 * ** + − = JacobiIdentity? JordanAlgebra? alternative? antiAssociative? antiCommutative? antiCommutator apply associative? associator associatorDependence basis coerce commutative? commutator conditionsForIdempotents convert coordinates elt flexible? jordanAdmissible? leftAlternative? leftCharacteristicPolynomial leftDiscriminant leftMinimalPolynomial leftNorm leftPower leftRankPolynomial leftRecip leftRegularRepresentation leftTrace leftTraceMatrix leftUnit leftUnits lieAdmissible? lieAlgebra? noncommutativeJordanAlgebra? plenaryPower powerAssociative? rank recip represents rightAlternative? rightCharacteristicPolynomial rightDiscriminant rightMinimalPolynomial rightNorm rightPower rightRankPolynomial rightRecip rightRegularRepresentation rightTrace rightTraceMatrix rightUnit rightUnits someBasis structuralConstants unit zero? ALGFF AlgebraicFunctionField FunctionFieldCategory 0 1 * ** + − / = D absolutelyIrreducible? associates? basis branchPoint? branchPointAtInfinity? character-

istic characteristicPolynomial charthRoot coerce complementaryBasis convert coordinates definingPolynomial derivationCoordinates differentiate discriminant divide elt euclideanSize expressIdealMember exquo extendedEuclidean factor gcd generator genus integral? integralAtInfinity? integralBasis integralBasisAtInfinity integralCoordinates integralDerivationMatrix integralMatrix integralMatrixAtInfinity integralRepresents inv inverseIntegralMatrix inverseIntegralMatrixAtInfinity knownInfBasis lcm lift minimalPolynomial multiEuclidean nonSingularModel norm normalizeAtInfinity numberOfComponents one? prime? primitivePart principalIdeal quo ramified? ramifiedAtInfinity? rank rationalPoint? rationalPoints recip reduce reduceBasisAtInfinity reducedSystem regularRepresentation rem represents retract retractIfCan singular? singularAtInfinity? sizeLess? squareFree squareFreePart trace traceMatrix unit? unitCanonical unitNormal yCoordinates zero? AN AlgebraicNumber AlgebraicallyClosedField CharacteristicZero ConvertibleTo DifferentialRing ExpressionSpace LinearlyExplicitRingOver RealConstant RetractableTo 0 1 * ** + - / < = D associates? belong? box characteristic coerce convert definingPolynomial denom differentiate distribute divide elt euclideanSize eval expressIdealMember exquo extendedEuclidean factor freeOf? gcd height inv is? kernel kernels lcm mainKernel map max min minPoly multiEuclidean nthRoot numer one? operator operators paren prime? principalIdeal quo recip reduce reducedSystem rem retract retractIfCan rootOf rootsOf sizeLess? sqrt squareFree squareFreePart subst tower unit? unitCanonical unitNormal zero? zeroOf zerosOf ANON AnonymousFunction SetCategory = coerce ANTISYM AntiSymm LeftAlgebra RetractableTo 0 1 * ** + - = characteristic coefficient coerce degree exp generator homogeneous? leadingBasisTerm leadingCoefficient map one? recip reductum retract retractIfCan retractable? zero? ANY Any SetCategory = any coerce domain domainOf obj objectOf showTypeInOutput ASTACK ArrayStack StackAggregate # = any? arrayStack bag coerce copy count depth empty empty? eq? every? extract! insert! inspect less? map map! member? members more? parts pop! push! size? top JORDAN AssociatedJordanAlgebra CoercibleTo FiniteRankNonAssociativeAlgebra FramedNonAssociativeAlgebra NonAssociativeAlgebra 0 * ** + - = JacobiIdentity? JordanAlgebra? alternative? antiAssociative? antiCommutative? antiCommutator apply associative? associator associatorDependence basis coerce commutative? commutator conditionsForIdempotents convert coordinates elt flexible? jordanAdmissible? leftAlternative? leftCharacteristicPolynomial leftDiscriminant leftMinimalPolynomial leftNorm leftPower leftRankPolynomial leftRecip leftRegularRepresentation leftTrace leftTraceMatrix leftUnit leftUnits lieAdmissible? lieAlgebra? noncommutativeJordanAlgebra? plenaryPower powerAssociative? rank recip represents rightAlternative? rightCharac-

teristicPolynomial rightDiscriminant rightMinimalPolynomial right-
Norm rightPower rightRankPolynomial rightRecip rightRegularRep-
resentation rightTrace rightTraceMatrix rightUnit rightUnits some-
Basis structuralConstants unit zero?   LIE AssociatedLieAlgebra
CoercibleTo FiniteRankNonAssociativeAlgebra FramedNonAssocia-
tiveAlgebra NonAssociativeAlgebra 0 ∗ ∗∗ + − = JacobiIdentity? Jor-
danAlgebra? alternative? antiAssociative? antiCommutative? an-
tiCommutator apply associative? associator associatorDependence
basis coerce commutative? commutator conditionsForIdempotents
convert coordinates elt flexible? jordanAdmissible? leftAlternative?
leftCharacteristicPolynomial leftDiscriminant leftMinimalPolynomial
leftNorm leftPower leftRankPolynomial leftRecip leftRegularRepre-
sentation leftTrace leftTraceMatrix leftUnit leftUnits lieAdmissible?
lieAlgebra?   noncommutativeJordanAlgebra?   plenaryPower pow-
erAssociative? rank recip represents rightAlternative? rightCharac-
teristicPolynomial rightDiscriminant rightMinimalPolynomial right-
Norm rightPower rightRankPolynomial rightRecip rightRegularRep-
resentation rightTrace rightTraceMatrix rightUnit rightUnits some-
Basis structuralConstants unit zero? ALIST AssociationList Associ-
ationListAggregate # = any? assoc bag child? children coerce con-
cat concat! construct copy copyInto! count cycleEntry cycleLength
cycleSplit! cycleTail cyclic? delete delete! dictionary distance elt
empty empty? entries entry? eq? every? explicitlyFinite? extract!
fill! find first index? indices insert insert! inspect key? keys last
leaf? less? list map map! maxIndex member? members merge
merge! minIndex more? new node? nodes parts position possi-
blyInfinite? qelt qsetelt! reduce remove remove! removeDuplicates
removeDuplicates! rest reverse reverse! search second select select!
setchildren! setelt setfirst! setlast! setrest! setvalue! size? sort
sort! sorted? split! swap! table tail third value BBTREE Bal-
ancedBinaryTree BinaryTreeCategory # = any? balancedBinaryTree
children coerce copy count cyclic? elt empty empty? eq? every?
leaf? leaves left less? map map! mapDown! mapUp! member?
members more? node node? nodes parts right setchildren! setelt
setleaves! setleft! setright! setvalue! size? value BPADIC Bal-
ancedPAdicInteger PAdicIntegerCategory 0 1 ∗ ∗∗ + − = approximate
associates? characteristic coerce complete digits divide euclidean-
Size expressIdealMember exquo extend extendedEuclidean gcd lcm
moduloP modulus multiEuclidean one? order principalIdeal quo quo-
tientByP recip rem sizeLess? sqrt unit? unitCanonical unitNormal
zero? BPADICRT BalancedPAdicRational QuotientFieldCategory 0
1 ∗ ∗∗ + − / = D approximate associates? characteristic coerce contin-
uedFraction denom denominator differentiate divide euclideanSize ex-
pressIdealMember exquo extendedEuclidean factor fractionPart gcd
inv lcm map multiEuclidean numer numerator one? prime? princi-
palIdeal quo recip reducedSystem rem removeZeroes retract retract-

IfCan sizeLess? squareFree squareFreePart unit? unitCanonical unit-Normal wholePart zero? BOP BasicOperator OrderedSet < = arity assert coerce comparison copy deleteProperty! display equality has? input is? max min name nary? nullary? operator properties property setProperties setProperty unary? weight BINARY BinaryExpansion QuotientFieldCategory 0 1 * ** + - / < = D abs associates? binary ceiling characteristic coerce convert denom denominator differentiate divide euclideanSize expressIdealMember exquo extendedEuclidean factor floor fractionPart gcd init inv lcm map max min multiEuclidean negative? nextItem numer numerator one? patternMatch positive? prime? principalIdeal quo random recip reducedSystem rem retract retractIfCan sign sizeLess? squareFree squareFreePart unit? unit-Canonical unitNormal wholePart zero? BSTREE BinarySearchTree BinaryTreeCategory # = any? binarySearchTree children coerce copy count cyclic? elt empty empty? eq? every? insert! insertRoot! leaf? leaves left less? map map! member? members more? node node? nodes parts right setchildren! setelt setleft! setright! setvalue! size? split value BTOURN BinaryTournament BinaryTreeCategory # = any? binaryTournament children coerce copy count cyclic? elt empty empty? eq? every? insert! leaf? leaves left less? map map! member? members more? node node? nodes parts right setchildren! setelt setleft! setright! setvalue! size? value BTREE BinaryTree BinaryTreeCategory # = any? binaryTree children coerce copy count cyclic? elt empty empty? eq? every? leaf? leaves left less? map map! member? members more? node node? nodes parts right setchildren! setelt setleft! setright! setvalue! size? value BITS Bits BitAggregate # < = ^ and any? bits coerce concat construct convert copy copyInto! count delete elt empty empty? entries entry? eq? every? fill! find first index? indices insert less? map map! max maxIndex member? members merge min minIndex more? nand new nor not or parts position qelt qsetelt! reduce remove removeDuplicates reverse reverse! select setelt size? sort sort! sorted? swap! xor BOOLEAN Boolean ConvertibleTo Finite OrderedSet < = ^ and coerce convert false implies index lookup max min nand nor not or random size true xor CARD CardinalNumber CancellationAbelianMonoid Monoid OrderedSet RetractableTo 0 1 * ** + - < = Aleph coerce countable? finite? generalizedContinuumHypothesisAssumed generalizedContinuumHypothesisAssumed? max min one? recip retract retractIfCan zero? CARTEN CartesianTensor GradedAlgebra 0 1 * + - = coerce contract degree elt kroneckerDelta leviCivitaSymbol product rank ravel reindex retract retractIfCan transpose unravel CCLASS CharacterClass ConvertibleTo FiniteSetAggregate SetCategory # < = alphabetic alphanumeric any? bag brace cardinality charClass coerce complement construct convert copy count dictionary difference digit empty empty? eq? every? extract! find hexDigit index insert! inspect intersect less? lookup lowerCase map

map! max member? members min more? parts random reduce remove remove! removeDuplicates select select! size size? subset? symmetricDifference union universe upperCase CHAR Character OrderedFinite < = alphabetic? alphanumeric? char coerce digit? escape hexDigit? index lookup lowerCase lowerCase? max min ord quote random size space upperCase upperCase? CLIF CliffordAlgebra Algebra Ring VectorSpace 0 1 * ** + − / = characteristic coefficient coerce dimension e monomial one? recip zero? COLOR Color AbelianSemiGroup * + = blue coerce color green hue numberOfHues red yellow COMM Commutator SetCategory = coerce mkcomm COMPLEX Complex ComplexCategory 0 1 * ** + − / < = D abs acos acosh acot acoth acsc acsch argument asec asech asin asinh associates? atan atanh basis characteristic characteristicPolynomial charthRoot coerce complex conditionP conjugate convert coordinates cos cosh cot coth createPrimitiveElement csc csch definingPolynomial derivationCoordinates differentiate discreteLog discriminant divide elt euclideanSize eval exp expressIdealMember exquo extendedEuclidean factor factorPolynomial factorSquareFreePolynomial factorsOfCyclicGroupSize gcd gcdPolynomial generator imag imaginary index init inv lcm lift log lookup map max min minimalPolynomial multiEuclidean nextItem norm nthRoot one? order pi polarCoordinates prime? primeFrobenius primitive? primitiveElement principalIdeal quo random rank rational rational? rationalIfCan real recip reduce reducedSystem regularRepresentation rem representationType represents retract retractIfCan sec sech sin sinh size sizeLess? solveLinearPolynomialEquation sqrt squareFree squareFreePart squareFreePolynomial tableForDiscreteLogarithm tan tanh trace traceMatrix unit? unitCanonical unitNormal zero? CONTFRAC ContinuedFraction Algebra Field 0 1 * ** + − / = approximants associates? characteristic coerce complete continuedFraction convergents denominators divide euclideanSize expressIdealMember exquo extend extendedEuclidean factor gcd inv lcm multiEuclidean numerators one? partialDenominators partialNumerators partialQuotients prime? principalIdeal quo recip reducedContinuedFraction reducedForm rem sizeLess? squareFree squareFreePart unit? unitCanonical unitNormal wholePart zero? DBASE Database SetCategory + − = coerce display elt fullDisplay DFLOAT DoubleFloat ConvertibleTo DifferentialRing FloatingPointSystem TranscendentalFunctionCategory 0 1 * ** + − / < = D abs acos acosh acot acoth acsc acsch asec asech asin asinh associates? atan atanh base bits ceiling characteristic coerce convert cos cosh cot coth csc csch decreasePrecision differentiate digits divide euclideanSize exp exp1 exponent expressIdealMember exquo extendedEuclidean factor float floor fractionPart gcd hash increasePrecision inv lcm log log10 log2 mantissa max min multiEuclidean negative? norm nthRoot one? order patternMatch pi positive? precision prime? principalIdeal quo rationalApproximation recip rem retract

retractIfCan round sec sech sign sin sinh sizeLess? sqrt squareFree squareFreePart tan tanh truncate unit? unitCanonical unitNormal wholePart zero? DLIST DataList ListAggregate # < = any? children coerce concat concat! construct convert copy copyInto! count cycleEntry cycleLength cycleSplit! cycleTail cyclic? datalist delete delete! elt empty empty? entries entry? eq? every? explicitlyFinite? fill! find first index? indices insert insert! last leaf? leaves less? list map map! max maxIndex member? members merge merge! min minIndex more? new node? nodes parts position possiblyInfinite? qelt qsetelt! reduce remove remove! removeDuplicates removeDuplicates! rest reverse reverse! second select select! setchildren! setelt setfirst! setlast! setrest! setvalue! size? sort sort! sorted? split! swap! tail third value DECIMAL DecimalExpansion QuotientFieldCategory 0 1 * ** + - / < = D abs associates? ceiling characteristic coerce convert decimal denom denominator differentiate divide euclideanSize expressIdealMember exquo extendedEuclidean factor floor fractionPart gcd init inv lcm map max min multiEuclidean negative? nextItem numer numerator one? patternMatch positive? prime? principalIdeal quo random recip reducedSystem rem retract retractIfCan sign sizeLess? squareFree squareFreePart unit? unitCanonical unitNormal wholePart zero? DHMATRIX DenavitHartenbergMatrix MatrixCategory # * ** + - / = antisymmetric? any? coerce column copy count determinant diagonal? diagonalMatrix elt empty empty? eq? every? exquo fill! horizConcat identity inverse less? listOfLists map map! matrix maxColIndex maxRowIndex member? members minColIndex minRowIndex minordet more? ncols new nrows nullSpace nullity parts qelt qsetelt! rank rotatex rotatey rotatez row rowEchelon scalarMatrix scale setColumn! setRow! setelt setsubMatrix! size? square? squareTop subMatrix swapColumns! swapRows! symmetric? translate transpose vertConcat zero DEQUEUE Dequeue DequeueAggregate # = any? back bag bottom! coerce copy count depth dequeue dequeue! empty empty? enqueue! eq? every? extract! extractBottom! extractTop! front height insert! insertBottom! insertTop! inspect length less? map map! member? members more? parts pop! push! reverse! rotate! size? top top! DERHAM DeRhamComplex LeftAlgebra RetractableTo 0 1 * ** + - = characteristic coefficient coerce degree exteriorDifferential generator homogeneous? leadingBasisTerm leadingCoefficient map one? recip reductum retract retractIfCan retractable? totalDifferential zero? DSMP DifferentialSparseMultivariatePolynomial DifferentialPolynomialCategory RetractableTo 0 1 * ** + - / < = D associates? characteristic charthRoot coefficient coefficients coerce conditionP content convert degree differentialVariables differentiate discriminant eval exquo factor factorPolynomial factorSquareFreePolynomial gcd gcdPolynomial ground ground? initial isExpt isPlus isTimes isobaric? lcm leader leadingCoefficient lead-

ingMonomial mainVariable makeVariable map mapExponents max min minimumDegree monicDivide monomial monomial? monomials multivariate numberOfMonomials one? order patternMatch prime? primitiveMonomials primitivePart recip reducedSystem reductum resultant retract retractIfCan separant solveLinearPolynomialEquation squareFree squareFreePart squareFreePolynomial totalDegree unit? unitCanonical unitNormal univariate variables weight weights zero? DPMM DirectProductMatrixModule DirectProductCategory LeftModule 0 1 # * ** + - / < = D abs any? characteristic coerce copy count differentiate dimension directProduct dot elt empty empty? entries entry? eq? every? fill! first index index? indices less? lookup map map! max maxIndex member? members min minIndex more? negative? one? parts positive? qelt qsetelt! random recip reducedSystem retract retractIfCan setelt sign size size? sup swap! unitVector zero? DPMO DirectProductModule DirectProductCategory LeftModule 0 1 # * ** + - / < = D abs any? characteristic coerce copy count differentiate dimension directProduct dot elt empty empty? entries entry? eq? every? fill! first index index? indices less? lookup map map! max maxIndex member? members min minIndex more? negative? one? parts positive? qelt qsetelt! random recip reducedSystem retract retractIfCan setelt sign size size? sup swap! unitVector zero? DIRPROD DirectProduct DirectProductCategory 0 1 # * ** + - / < = D abs any? characteristic coerce copy count differentiate dimension directProduct dot elt empty empty? entries entry? eq? every? fill! first index index? indices less? lookup map map! max maxIndex member? members min minIndex more? negative? one? parts positive? qelt qsetelt! random recip reducedSystem retract retractIfCan setelt sign size size? sup swap! unitVector zero? DMP DistributedMultivariatePolynomial PolynomialCategory 0 1 * ** + - / < = D associates? characteristic charthRoot coefficient coefficients coerce conditionP const content convert degree differentiate discriminant eval exquo factor factorPolynomial factorSquareFreePolynomial gcd gcdPolynomial ground ground? isExpt isPlus isTimes lcm leadingCoefficient leadingMonomial mainVariable map mapExponents max min minimumDegree monicDivide monomial monomial? monomials multivariate numberOfMonomials one? prime? primitiveMonomials primitivePart recip reducedSystem reductum reorder resultant retract retractIfCan solveLinearPolynomialEquation squareFree squareFreePart squareFreePolynomial totalDegree unit? unitCanonical unitNormal univariate variables zero? DROPT DrawOption SetCategory = adaptive clip coerce colorFunction coordinate coordinates curveColor option option? pointColor range ranges space style title toScale tubePoints tubeRadius unit var1Steps var2Steps EFULS ElementaryFunctionsUnivariateLaurentSeries PartialTranscendentalFunctions ** acos acosIfCan acosh acoshIfCan acot acotIfCan acoth acothIfCan acsc acscIfCan acsch acschIfCan asec asecIfCan asech

asechIfCan asin asinIfCan asinh asinhIfCan atan atanIfCan atanh atanhIfCan cos cosIfCan cosh coshIfCan cot cotIfCan coth cothIfCan csc cscIfCan csch cschIfCan exp expIfCan log logIfCan nthRootIfCan sec secIfCan sech sechIfCan sin sinIfCan sinh sinhIfCan tan tanIfCan tanh tanhIfCan EFUPXS ElementaryFunctionsUnivariatePuiseuxSeries PartialTranscendentalFunctions ** acos acosIfCan acosh acoshIfCan acot acotIfCan acoth acothIfCan acsc acscIfCan acsch acschIfCan asec asecIfCan asech asechIfCan asin asinIfCan asinh asinhIfCan atan atanIfCan atanh atanhIfCan cos cosIfCan cosh coshIfCan cot cotIfCan coth cothIfCan csc cscIfCan csch cschIfCan exp expIfCan log logIfCan nthRootIfCan sec secIfCan sech sechIfCan sin sinIfCan sinh sinhIfCan tan tanIfCan tanh tanhIfCan EQTBL EqTable TableAggregate # = any? bag coerce construct copy count dictionary elt empty empty? entries entry? eq? every? extract! fill! find first index? indices insert! inspect key? keys less? map map! maxIndex member? members minIndex more? parts qelt qsetelt! reduce remove remove! removeDuplicates search select select! setelt size? swap! table EQ Equation CoercibleTo InnerEvalable Object SetCategory * ** + - = coerce equation eval lhs map rhs EMR EuclideanModularRing EuclideanDomain 0 1 * ** + - = associates? characteristic coerce divide euclideanSize exQuo expressIdealMember exquo extendedEuclidean gcd inv lcm modulus multiEuclidean one? principalIdeal quo recip reduce rem sizeLess? unit? unitCanonical unitNormal zero? EXIT Exit SetCategory = coerce EXPR Expression AlgebraicallyClosedFunctionSpace CombinatorialOpsCategory FunctionSpace LiouvillianFunctionCategory RetractableTo SpecialFunctionCategory TranscendentalFunctionCategory 0 1 * ** + - / < = Beta Ci D Ei Gamma Si abs acos acosh acot acoth acsc acsch airyAi airyBi applyQuote asec asech asin asinh associates? atan atanh belong? besselI besselJ besselK besselY binomial box characteristic charthRoot coerce commutator conjugate convert cos cosh cot coth csc csch definingPolynomial denom denominator differentiate digamma dilog distribute divide elt erf euclideanSize eval exp expressIdealMember exquo extendedEuclidean factor factorial factorials freeOf? gcd ground ground? height integral inv is? isExpt isMult isPlus isPower isTimes kernel kernels lcm li log mainKernel map max min minPoly multiEuclidean nthRoot numer numerator one? operator operators paren patternMatch permutation pi polygamma prime? principalIdeal product quo recip reduce reducedSystem rem retract retractIfCan rootOf rootsOf sec sech sin sinh sizeLess? sqrt squareFree squareFreePart subst summation tan tanh tower unit? unitCanonical unitNormal univariate variables zero? zeroOf zerosOf EAB ExtAlgBasis OrderedSet < = Nul coerce degree exponents max min FR Factored Algebra DifferentialExtension Eltable Evalable FullyEvalableOver FullyRetractableTo GcdDomain InnerEvalable IntegralDomain RealConstant UniqueFactorizationDomain 0 1 * ** + - = D

associates? characteristic coerce convert differentiate elt eval expand exponent exquo factor factorList factors flagFactor gcd irreducibleFactor lcm makeFR map nilFactor nthExponent nthFactor nthFlag numberOfFactors one? prime? primeFactor rational rational? rationalIfCan recip retract retractIfCan sqfrFactor squareFree squareFreePart unit unit? unitCanonical unitNormal unitNormalize zero? FNAME FileName FileNameCategory = coerce directory exists? extension filename name new readable? writable? FILE File FileCategory = close! coerce iomode name open read! readIfCan! reopen! write! FDIV FiniteDivisor AbelianGroup 0 * + - = algsplit coerce divisor finiteBasis generator ideal lSpaceBasis mkBasicDiv principal? reduce zero? FFCGP FiniteFieldCyclicGroupExtensionByPolynomial FiniteAlgebraicExtensionField 0 1 * ** + - / = Frobenius algebraic? associates? basis characteristic charthRoot coerce conditionP coordinates createNormalElement createPrimitiveElement definingPolynomial degree dimension discreteLog divide euclideanSize expressIdealMember exquo extendedEuclidean extensionDegree factor factorsOfCyclicGroupSize gcd generator getZechTable inGroundField? index init inv lcm lookup minimalPolynomial multiEuclidean nextItem norm normal? normalElement one? order prime? primeFrobenius primitive? primitiveElement principalIdeal quo random recip rem representationType represents retract retractIfCan size sizeLess? squareFree squareFreePart tableForDiscreteLogarithm trace transcendenceDegree transcendent? unit? unitCanonical unitNormal zero? FFCGX FiniteFieldCyclicGroupExtension FiniteAlgebraicExtensionField 0 1 * ** + - / = Frobenius algebraic? associates? basis characteristic charthRoot coerce conditionP coordinates createNormalElement createPrimitiveElement definingPolynomial degree dimension discreteLog divide euclideanSize expressIdealMember exquo extendedEuclidean extensionDegree factor factorsOfCyclicGroupSize gcd generator getZechTable inGroundField? index init inv lcm lookup minimalPolynomial multiEuclidean nextItem norm normal? normalElement one? order prime? primeFrobenius primitive? primitiveElement principalIdeal quo random recip rem representationType represents retract retractIfCan size sizeLess? squareFree squareFreePart tableForDiscreteLogarithm trace transcendenceDegree transcendent? unit? unitCanonical unitNormal zero? FFCG FiniteFieldCyclicGroup FiniteAlgebraicExtensionField 0 1 * ** + - / = Frobenius algebraic? associates? basis characteristic charthRoot coerce conditionP coordinates createNormalElement createPrimitiveElement definingPolynomial degree dimension discreteLog divide euclideanSize expressIdealMember exquo extendedEuclidean extensionDegree factor factorsOfCyclicGroupSize gcd generator getZechTable inGroundField? index init inv lcm lookup minimalPolynomial multiEuclidean nextItem norm normal? normalElement one? order prime? primeFrobenius primitive? primitiveElement princi-

palIdeal quo random recip rem representationType represents retract retractIfCan size sizeLess? squareFree squareFreePart tableForDiscreteLogarithm trace transcendenceDegree transcendent? unit? unitCanonical unitNormal zero? FFP FiniteFieldExtensionByPolynomial FiniteAlgebraicExtensionField 0 1 * ** + - / = Frobenius algebraic? associates? basis characteristic charthRoot coerce conditionP coordinates createNormalElement createPrimitiveElement definingPolynomial degree dimension discreteLog divide euclideanSize expressIdealMember exquo extendedEuclidean extensionDegree factor factorsOfCyclicGroupSize gcd generator inGroundField? index init inv lcm lookup minimalPolynomial multiEuclidean nextItem norm normal? normalElement one? order prime? primeFrobenius primitive? primitiveElement principalIdeal quo random recip rem representationType represents retract retractIfCan size sizeLess? squareFree squareFreePart tableForDiscreteLogarithm trace transcendenceDegree transcendent? unit? unitCanonical unitNormal zero? FFX FiniteFieldExtension FiniteAlgebraicExtensionField 0 1 * ** + - / = Frobenius algebraic? associates? basis characteristic charthRoot coerce conditionP coordinates createNormalElement createPrimitiveElement definingPolynomial degree dimension discreteLog divide euclideanSize expressIdealMember exquo extendedEuclidean extensionDegree factor factorsOfCyclicGroupSize gcd generator inGroundField? index init inv lcm lookup minimalPolynomial multiEuclidean nextItem norm normal? normalElement one? order prime? primeFrobenius primitive? primitiveElement principalIdeal quo random recip rem representationType represents retract retractIfCan size sizeLess? squareFree squareFreePart tableForDiscreteLogarithm trace transcendenceDegree transcendent? unit? unitCanonical unitNormal zero? FFNBP FiniteFieldNormalBasisExtensionByPolynomial FiniteAlgebraicExtensionField 0 1 * ** + - / = Frobenius algebraic? associates? basis characteristic charthRoot coerce conditionP coordinates createNormalElement createPrimitiveElement definingPolynomial degree dimension discreteLog divide euclideanSize expressIdealMember exquo extendedEuclidean extensionDegree factor factorsOfCyclicGroupSize gcd generator getMultiplicationMatrix getMultiplicationTable inGroundField? index init inv lcm lookup minimalPolynomial multiEuclidean nextItem norm normal? normalElement one? order prime? primeFrobenius primitive? primitiveElement principalIdeal quo random recip rem representationType represents retract retractIfCan size sizeLess? sizeMultiplication squareFree squareFreePart tableForDiscreteLogarithm trace transcendenceDegree transcendent? unit? unitCanonical unitNormal zero? FFNBX FiniteFieldNormalBasisExtension FiniteAlgebraicExtensionField 0 1 * ** + - / = Frobenius algebraic? associates? basis characteristic charthRoot coerce conditionP coordinates createNormalElement createPrimitiveElement defining-

Polynomial degree dimension discreteLog divide euclideanSize expressIdealMember exquo extendedEuclidean extensionDegree factor factorsOfCyclicGroupSize gcd generator getMultiplicationMatrix getMultiplicationTable inGroundField? index init inv lcm lookup minimalPolynomial multiEuclidean nextItem norm normal? normalElement one? order prime? primeFrobenius primitive? primitiveElement principalIdeal quo random recip rem representationType represents retract retractIfCan size sizeLess? sizeMultiplication squareFree squareFreePart tableForDiscreteLogarithm trace transcendenceDegree transcendent? unit? unitCanonical unitNormal zero? FFNB FiniteFieldNormalBasis FiniteAlgebraicExtensionField 0 1 * ** + − / = Frobenius algebraic? associates? basis characteristic charthRoot coerce conditionP coordinates createNormalElement createPrimitiveElement definingPolynomial degree dimension discreteLog divide euclideanSize expressIdealMember exquo extendedEuclidean extensionDegree factor factorsOfCyclicGroupSize gcd generator getMultiplicationMatrix getMultiplicationTable inGroundField? index init inv lcm lookup minimalPolynomial multiEuclidean nextItem norm normal? normalElement one? order prime? primeFrobenius primitive? primitiveElement principalIdeal quo random recip rem representationType represents retract retractIfCan size sizeLess? sizeMultiplication squareFree squareFreePart tableForDiscreteLogarithm trace transcendenceDegree transcendent? unit? unitCanonical unitNormal zero? FF FiniteField FiniteAlgebraicExtensionField 0 1 * ** + − / = Frobenius algebraic? associates? basis characteristic charthRoot coerce conditionP coordinates createNormalElement createPrimitiveElement definingPolynomial degree dimension discreteLog divide euclideanSize expressIdealMember exquo extendedEuclidean extensionDegree factor factorsOfCyclicGroupSize gcd generator inGroundField? index init inv lcm lookup minimalPolynomial multiEuclidean nextItem norm normal? normalElement one? order prime? primeFrobenius primitive? primitiveElement principalIdeal quo random recip rem representationType represents retract retractIfCan size sizeLess? squareFree squareFreePart tableForDiscreteLogarithm trace transcendenceDegree transcendent? unit? unitCanonical unitNormal zero? FARRAY FlexibleArray ExtensibleLinearAggregate OneDimensionalArrayAggregate # < = any? coerce concat concat! construct convert copy copyInto! count delete delete! elt empty empty? entries entry? eq? every? fill! find first flexibleArray index? indices insert insert! less? map map! max maxIndex member? members merge merge! min minIndex more? new parts physicalLength physicalLength! position qelt qsetelt! reduce remove remove! removeDuplicates removeDuplicates! reverse reverse! select select! setelt shrinkable size? sort sort! sorted? swap! FLOAT Float CoercibleTo ConvertibleTo DifferentialRing FloatingPointSystem TranscendentalFunctionCategory 0 1 * **

+ − / < = D abs acos acosh acot acoth acsc acsch asec asech asin asinh associates? atan atanh base bits ceiling characteristic coerce convert cos cosh cot coth csc csch decreasePrecision differentiate digits divide euclideanSize exp exp1 exponent expressIdealMember exquo extendedEuclidean factor float floor fractionPart gcd increasePrecision inv lcm log log10 log2 mantissa max min multiEuclidean negative? norm normalize nthRoot one? order outputFixed outputFloating outputGeneral outputSpacing patternMatch pi positive? precision prime? principalIdeal quo rationalApproximation recip relerror rem retract retractIfCan round sec sech shift sign sin sinh sizeLess? sqrt squareFree squareFreePart tan tanh truncate unit? unitCanonical unitNormal wholePart zero? **FRIDEAL FractionalIdeal Group 1 ∗ ∗∗ / =** basis coerce commutator conjugate denom ideal inv minimize norm numer one? randomLC recip **FRAC Fraction QuotientFieldCategory 0 1 ∗ ∗∗ + − / < = D** abs associates? ceiling characteristic charthRoot coerce conditionP convert denom denominator differentiate divide elt euclideanSize eval expressIdealMember exquo extendedEuclidean factor factorPolynomial factorSquareFreePolynomial floor fractionPart gcd gcdPolynomial init inv lcm map max min multiEuclidean negative? nextItem numer numerator one? patternMatch positive? prime? principalIdeal quo random recip reducedSystem rem retract retractIfCan sign sizeLess? solveLinearPolynomialEquation squareFree squareFreePart squareFreePolynomial unit? unitCanonical unitNormal wholePart zero? **FRMOD FramedModule Monoid 1 ∗ ∗∗ =** basis coerce module norm one? recip **FAGROUP FreeAbelianGroup AbelianGroup FreeAbelianMonoidCategory Module OrderedSet 0 ∗ + − < =** coefficient coerce highCommonTerms mapCoef mapGen max min nthCoef nthFactor retract retractIfCan size terms zero? **FAMONOID FreeAbelianMonoid FreeAbelianMonoidCategory 0 ∗ + − =** coefficient coerce highCommonTerms mapCoef mapGen nthCoef nthFactor retract retractIfCan size terms zero? **FGROUP FreeGroup Group RetractableTo 1 ∗ ∗∗ / =** coerce commutator conjugate factors inv mapExpon mapGen nthExpon nthFactor one? recip retract retractIfCan size **FM FreeModule BiModule IndexedDirectProductCategory Module 0 ∗ + − =** coerce leadingCoefficient leadingSupport map monomial reductum zero? **FMONOID FreeMonoid Monoid OrderedSet RetractableTo 1 ∗ ∗∗ < =** coerce divide factors hclf hcrf lquo mapExpon mapGen max min nthExpon nthFactor one? overlap recip retract retractIfCan rquo size **FNLA FreeNilpotentLie NonAssociativeAlgebra 0 ∗ ∗∗ + − =** antiCommutator associator coerce commutator deepExpand dimension generator leftPower rightPower shallowExpand zero? **FUNCTION FunctionCalled SetCategory =** coerce name **GDMP GeneralDistributedMultivariatePolynomial PolynomialCategory 0 1 ∗ ∗∗ + − / < = D** associates? characteristic charthRoot coefficient coefficients coerce conditionP const content convert degree differentiate discriminant eval exquo factor factorPolynomial

factorSquareFreePolynomial gcd gcdPolynomial ground ground? isExpt isPlus isTimes lcm leadingCoefficient leadingMonomial mainVariable map mapExponents max min minimumDegree monicDivide monomial monomial? monomials multivariate numberOfMonomials one? prime? primitiveMonomials primitivePart recip reducedSystem reductum reorder resultant retract retractIfCan solveLinearPolynomialEquation squareFree squareFreePart squareFreePolynomial totalDegree unit? unitCanonical unitNormal univariate variables zero? GSTBL GeneralSparseTable TableAggregate # = any? bag coerce construct copy count dictionary elt empty empty? entries entry? eq? every? extract! fill! find first index? indices insert! inspect key? keys less? map map! maxIndex member? members minIndex more? parts qelt qsetelt! reduce remove remove! removeDuplicates search select select! setelt size? swap! table GCNAALG GenericNonAssociativeAlgebra FramedNonAssociativeAlgebra LeftModule 0 * ** + - = JacobiIdentity? JordanAlgebra? alternative? antiAssociative? antiCommutative? antiCommutator apply associative? associator associatorDependence basis coerce commutative? commutator conditionsForIdempotents convert coordinates elt flexible? generic genericLeftDiscriminant genericLeftMinimalPolynomial genericLeftNorm genericLeftTrace genericLeftTraceForm genericRightDiscriminant genericRightMinimalPolynomial genericRightNorm genericRightTrace genericRightTraceForm jordanAdmissible? leftAlternative? leftCharacteristicPolynomial leftDiscriminant leftMinimalPolynomial leftNorm leftPower leftRankPolynomial leftRecip leftRegularRepresentation leftTrace leftTraceMatrix leftUnit leftUnits lieAdmissible? lieAlgebra? noncommutativeJordanAlgebra? plenaryPower powerAssociative? rank recip represents rightAlternative? rightCharacteristicPolynomial rightDiscriminant rightMinimalPolynomial rightNorm rightPower rightRankPolynomial rightRecip rightRegularRepresentation rightTrace rightTraceMatrix rightUnit rightUnits someBasis structuralConstants unit zero? GRIMAGE GraphImage SetCategory = appendPoint coerce component graphImage key makeGraphImage point pointLists putColorInfo ranges units HASHTBL HashTable TableAggregate # = any? bag coerce construct copy count dictionary elt empty empty? entries entry? eq? every? extract! fill! find first index? indices insert! inspect key? keys less? map map! maxIndex member? members minIndex more? parts qelt qsetelt! reduce remove remove! removeDuplicates search select select! setelt size? swap! table HEAP Heap PriorityQueueAggregate # = any? bag coerce copy count empty empty? eq? every? extract! heap insert! inspect less? map map! max member? members merge merge! more? parts size? HEXADEC HexadecimalExpansion QuotientFieldCategory 0 1 * ** + - / < = D abs associates? ceiling characteristic coerce convert denom denominator differentiate divide euclideanSize expressIdealMember exquo extendedEuclidean factor

floor fractionPart gcd hex init inv lcm map max min multiEuclidean negative? nextItem numer numerator one? patternMatch positive? prime? principalIdeal quo random recip reducedSystem rem retract retractIfCan sign sizeLess? squareFree squareFreePart unit? unitCanonical unitNormal wholePart zero? ICARD IndexCard OrderedSet < = coerce display elt fullDisplay max min IBITS IndexedBits BitAggregate # < = And Not Or ^ and any? coerce concat construct convert copy copyInto! count delete elt empty empty? entries entry? eq? every? fill! find first index? indices insert less? map map! max maxIndex member? members merge min minIndex more? nand new nor not or parts position qelt qsetelt! reduce remove removeDuplicates reverse reverse! select setelt size? sort sort! sorted? swap! xor IDPAG IndexedDirectProductAbelianGroup AbelianGroup IndexedDirectProductCategory 0 * + - = coerce leadingCoefficient leadingSupport map monomial reductum zero? IDPAM IndexedDirectProductAbelianMonoid AbelianMonoid IndexedDirectProductCategory 0 * + = coerce leadingCoefficient leadingSupport map monomial reductum zero? IDPO IndexedDirectProductObject IndexedDirectProductCategory = coerce leadingCoefficient leadingSupport map monomial reductum IDPOAMS IndexedDirectProductOrderedAbelianMonoidSup IndexedDirectProductCategory OrderedAbelianMonoidSup 0 * + - < = coerce leadingCoefficient leadingSupport map max min monomial reductum sup zero? IDPOAM IndexedDirectProductOrderedAbelianMonoid IndexedDirectProductCategory OrderedAbelianMonoid 0 * + < = coerce leadingCoefficient leadingSupport map max min monomial reductum zero? INDE IndexedExponents IndexedDirectProductCategory OrderedAbelianMonoidSup 0 * + - < = coerce leadingCoefficient leadingSupport map max min monomial reductum sup zero? IFARRAY IndexedFlexibleArray ExtensibleLinearAggregate OneDimensionalArrayAggregate # < = any? coerce concat concat! construct convert copy copyInto! count delete delete! elt empty empty? entries entry? eq? every? fill! find first flexibleArray index? indices insert insert! less? map map! max maxIndex member? members merge merge! min minIndex more? new parts physicalLength physicalLength! position qelt qsetelt! reduce remove remove! removeDuplicates removeDuplicates! reverse reverse! select select! setelt shrinkable size? sort sort! sorted? swap! ILIST IndexedList ListAggregate # < = any? child? children coerce concat concat! construct convert copy copyInto! count cycleEntry cycleLength cycleSplit! cycleTail cyclic? delete delete! distance elt empty empty? entries entry? eq? every? explicitlyFinite? fill! find first index? indices insert insert! last leaf? less? list map map! max maxIndex member? members merge merge! min minIndex more? new node? nodes parts position possiblyInfinite? qelt qsetelt! reduce remove remove! removeDuplicates removeDuplicates! rest reverse reverse! second select select! setchildren! setelt

setfirst! setlast! setrest! setvalue! size? sort sort! sorted? split! swap! tail third value IMATRIX IndexedMatrix MatrixCategory # ∗ ∗∗ + − / = antisymmetric? any? coerce column copy count determinant diagonal? diagonalMatrix elt empty empty? eq? every? exquo fill! horizConcat inverse less? listOfLists map map! matrix maxColIndex maxRowIndex member? members minColIndex minRowIndex minordet more? ncols new nrows nullSpace nullity parts qelt qsetelt! rank row rowEchelon scalarMatrix setColumn! setRow! setelt setsubMatrix! size? square? squareTop subMatrix swapColumns! swapRows! symmetric? transpose vertConcat zero IARRAY1 IndexedOneDimensionalArray OneDimensionalArrayAggregate # < = any? coerce concat construct convert copy copyInto! count delete elt empty empty? entries entry? eq? every? fill! find first index? indices insert less? map map! max maxIndex member? members merge min minIndex more? new parts position qelt qsetelt! reduce remove removeDuplicates reverse reverse! select setelt size? sort sort! sorted? swap! ISTRING IndexedString StringAggregate # < = any? coerce concat construct copy copyInto! count delete elt empty empty? entries entry? eq? every? fill! find first hash index? indices insert leftTrim less? lowerCase lowerCase! map map! match? max maxIndex member? members merge min minIndex more? new parts position prefix? qelt qsetelt! reduce remove removeDuplicates replace reverse reverse! rightTrim select setelt size? sort sort! sorted? split substring? suffix? swap! trim upperCase upperCase! IARRAY2 IndexedTwoDimensionalArray TwoDimensionalArrayCategory # = any? coerce column copy count elt empty empty? eq? every? fill! less? map map! maxColIndex maxRowIndex member? members minColIndex minRowIndex more? ncols new nrows parts qelt qsetelt! row setColumn! setRow! setelt size? IVECTOR IndexedVector VectorCategory # ∗ + − < = any? coerce concat construct convert copy copyInto! count delete dot elt empty empty? entries entry? eq? every? fill! find first index? indices insert less? map map! max maxIndex member? members merge min minIndex more? new parts position qelt qsetelt! reduce remove removeDuplicates reverse reverse! select setelt size? sort sort! sorted? swap! zero ITUPLE InfiniteTuple CoercibleTo coerce construct filterUntil filterWhile generate map select IFF InnerFiniteField FiniteAlgebraicExtensionField 0 1 ∗ ∗∗ + − / = Frobenius algebraic? associates? basis characteristic charthRoot coerce conditionP coordinates createNormalElement createPrimitiveElement definingPolynomial degree dimension discreteLog divide euclideanSize expressIdealMember exquo extendedEuclidean extensionDegree factor factorsOfCyclicGroupSize gcd generator inGroundField? index init inv lcm lookup minimalPolynomial multiEuclidean nextItem norm normal? normalElement one? order prime? primeFrobenius primitive? primitiveElement principalIdeal quo random recip rem representationType rep-

resents retract retractIfCan size sizeLess? squareFree squareFreePart
tableForDiscreteLogarithm trace transcendenceDegree transcendent?
unit? unitCanonical unitNormal zero? IFAMON InnerFreeAbelian-
Monoid FreeAbelianMonoidCategory 0 * + - = coefficient coerce high-
CommonTerms mapCoef mapGen nthCoef nthFactor retract retract-
IfCan size terms zero? IIARRAY2 InnerIndexedTwoDimensionalAr-
ray TwoDimensionalArrayCategory # = any? coerce column copy
count elt empty empty? eq? every? fill! less? map map! maxCol-
Index maxRowIndex member? members minColIndex minRowIndex
more? ncols new nrows parts qelt qsetelt! row setColumn! setRow!
setelt size? IPADIC InnerPAdicInteger PAdicIntegerCategory 0 1 *
** + - = approximate associates? characteristic coerce complete digits
divide euclideanSize expressIdealMember exquo extend extendedEu-
clidean gcd lcm moduloP modulus multiEuclidean one? order princi-
palIdeal quo quotientByP recip rem sizeLess? sqrt unit? unitCanon-
ical unitNormal zero? IPF InnerPrimeField ConvertibleTo FiniteAl-
gebraicExtensionField FiniteFieldCategory 0 1 * ** + - / = Frobe-
nius algebraic? associates? basis characteristic charthRoot coerce
conditionP convert coordinates createNormalElement createPrimi-
tiveElement definingPolynomial degree dimension discreteLog divide
euclideanSize expressIdealMember exquo extendedEuclidean exten-
sionDegree factor factorsOfCyclicGroupSize gcd generator inGround-
Field? index init inv lcm lookup minimalPolynomial multiEuclidean
nextItem norm normal? normalElement one? order prime? prime-
Frobenius primitive? primitiveElement principalIdeal quo random re-
cip rem representationType represents retract retractIfCan size size-
Less? squareFree squareFreePart tableForDiscreteLogarithm trace
transcendenceDegree transcendent? unit? unitCanonical unitNor-
mal zero? ITAYLOR InnerTaylorSeries IntegralDomain Ring 0 1 * **
+ - = associates? characteristic coefficients coerce exquo one? order
pole? recip series unit? unitCanonical unitNormal zero? INFORM
InputForm ConvertibleTo SExpressionCategory 0 1 # * ** + / = atom?
binary car cdr coerce compile convert declare destruct elt eq expr flat-
ten float float? function integer integer? interpret lambda list? null?
pair? string string? symbol symbol? uequal unparse ZMOD Inte-
gerMod CommutativeRing ConvertibleTo Finite StepThrough 0 1 *
** + - = characteristic coerce convert index init lookup nextItem one?
random recip size zero? INT Integer ConvertibleTo IntegerNumber-
System 0 1 * ** + - < = D abs addmod associates? base binomial bit?
characteristic coerce convert copy dec differentiate divide euclidean-
Size even? expressIdealMember exquo extendedEuclidean factor fac-
torial gcd hash inc init invmod lcm length mask max min mulmod
multiEuclidean negative? nextItem odd? one? patternMatch permu-
tation positive? positiveRemainder powmod prime? principalIdeal
quo random rational rational? rationalIfCan recip reducedSystem
rem retract retractIfCan shift sign sizeLess? squareFree square-

FreePart submod symmetricRemainder unit? unitCanonical unit-Normal zero? IR IntegrationResult Module RetractableTo 0 ∗ + − = D coerce differentiate elem? integral logpart mkAnswer notelem rat-part retract retractIfCan zero? KERNEL Kernel CachableSet ConvertibleTo Patternable < = argument coerce convert height is? kernel max min name operator position setPosition symbolIfCan KAFILE KeyedAccessFile FileCategory TableAggregate # = any? bag close! coerce construct copy count dictionary elt empty empty? entries entry? eq? every? extract! fill! find first index? indices in-sert! inspect iomode key? keys less? map map! maxIndex mem-ber? members minIndex more? name open pack! parts qelt qsetelt! read! reduce remove remove! removeDuplicates reopen! search select select! setelt size? swap! table write! LAUPOL Laurent-Polynomial CharacteristicNonZero CharacteristicZero ConvertibleTo DifferentialExtension EuclideanDomain FullyRetractableTo Integral-Domain RetractableTo 0 1 ∗ ∗∗ + − = D associates? characteristic charthRoot coefficient coerce convert degree differentiate divide eu-clideanSize expressIdealMember exquo extendedEuclidean gcd lcm leadingCoefficient monomial monomial? multiEuclidean one? order principalIdeal quo recip reductum rem retract retractIfCan separate sizeLess? trailingCoefficient unit? unitCanonical unitNormal zero? LIB Library TableAggregate # = any? bag coerce construct copy count dictionary elt empty empty? entries entry? eq? every? ex-tract! fill! find first index? indices insert! inspect key? keys less? library map map! maxIndex member? members minIndex more? pack! parts qelt qsetelt! reduce remove remove! removeDuplicates search select select! setelt size? swap! table LSQM LieSquareMa-trix CoercibleTo FramedNonAssociativeAlgebra SquareMatrixCate-gory 0 1 # ∗ ∗∗ + − / = D JacobiIdentity? JordanAlgebra? alter-native? antiAssociative? antiCommutative? antiCommutator an-tisymmetric? any? apply associative? associator associatorDepen-dence basis characteristic coerce column commutative? commutator conditionsForIdempotents convert coordinates copy count determi-nant diagonal diagonal? diagonalMatrix diagonalProduct differenti-ate elt empty empty? eq? every? exquo flexible? inverse jordanAd-missible? leftAlternative? leftCharacteristicPolynomial leftDiscrimi-nant leftMinimalPolynomial leftNorm leftPower leftRankPolynomial leftRecip leftRegularRepresentation leftTrace leftTraceMatrix left-Unit leftUnits less? lieAdmissible? lieAlgebra? listOfLists map map! matrix maxColIndex maxRowIndex member? members min-ColIndex minRowIndex minordet more? ncols noncommutativeJor-danAlgebra? nrows nullSpace nullity one? parts plenaryPower pow-erAssociative? qelt rank recip reducedSystem represents retract retractIfCan rightAlternative? rightCharacteristicPolynomial right-Discriminant rightMinimalPolynomial rightNorm rightPower righ-tRankPolynomial rightRecip rightRegularRepresentation rightTrace

rightTraceMatrix rightUnit rightUnits row rowEchelon scalarMatrix size? someBasis square? structuralConstants symmetric? trace unit zero? LODO LinearOrdinaryDifferentialOperator MonogenicLinear-Operator 0 1 * ** + − = D characteristic coefficient coerce degree elt leadingCoefficient leftDivide leftExactQuotient leftGcd leftLcm left-Quotient leftRemainder minimumDegree monomial one? recip reductum rightDivide rightExactQuotient rightGcd rightLcm rightQuotient rightRemainder zero? LMOPS ListMonoidOps RetractableTo SetCategory = coerce leftMult listOfMonoms makeMulti makeTerm makeUnit mapExpon mapGen nthExpon nthFactor outputForm plus retract retractIfCan reverse reverse! rightMult size LMDICT List-MultiDictionary MultiDictionary # = any? bag coerce construct convert copy count dictionary duplicates duplicates? empty empty? eq? every? extract! find insert! inspect less? map map! member? members more? parts reduce remove remove! removeDuplicates removeDuplicates! select select! size? substitute LIST List ListAggregate # < = any? append child? children coerce concat concat! cons construct convert copy copyInto! count cycleEntry cycleLength cycleSplit! cycleTail cyclic? delete delete! distance elt empty empty? entries entry? eq? every? explicitlyFinite? fill! find first index? indices insert insert! last leaf? less? list map map! max maxIndex member? members merge merge! min minIndex more? new nil node? nodes null parts position possiblyInfinite? qelt qsetelt! reduce remove remove! removeDuplicates removeDuplicates! rest reverse reverse! second select select! setDifference setIntersection setUnion setchildren! setelt setfirst! setlast! setrest! setvalue! size? sort sort! sorted? split! swap! tail third value LA LocalAlgebra Algebra OrderedRing 0 1 * ** + − / < = abs characteristic coerce denom max min negative? numer one? positive? recip sign zero? LO Localize Module OrderedAbelianGroup 0 * + − / < = coerce denom max min numer zero? MKCHSET MakeCachableSet CachableSet CoercibleTo < = coerce max min position setPosition MKODRING MakeOrdinaryDifferentialRing CoercibleTo DifferentialRing 0 1 * ** + − = D characteristic coerce differentiate one? recip zero? MATRIX Matrix MatrixCategory # * ** + − / = antisymmetric? any? coerce column copy count determinant diagonal? diagonalMatrix elt empty empty? eq? every? exquo fill! horizConcat inverse less? listOfLists map map! matrix maxColIndex maxRowIndex member? members minColIndex minRowIndex minordet more? ncols new nrows nullSpace nullity parts qelt qsetelt! rank row rowEchelon scalarMatrix setColumn! setRow! setelt setsubMatrix! size? square? squareTop subMatrix swapColumns! swapRows! symmetric? transpose vertConcat zero MODMON ModMonic Finite UnivariatePolynomialCategory 0 1 * ** + − / < = An D UnVectorise Vectorise associates? characteristic charthRoot coefficient coefficients coerce composite computePowers conditionP content degree differentiate discriminant divide divide-

Exponents elt euclideanSize eval expressIdealMember exquo extendedEuclidean factor factorPolynomial factorSquareFreePolynomial gcd gcdPolynomial ground ground? index init integrate isExpt isPlus isTimes lcm leadingCoefficient leadingMonomial lift lookup mainVariable makeSUP map mapExponents max min minimumDegree modulus monicDivide monomial monomial? monomials multiEuclidean multiplyExponents multivariate nextItem numberOfMonomials one? order pow prime? primitiveMonomials primitivePart principalIdeal pseudoDivide pseudoQuotient pseudoRemainder quo random recip reduce reducedSystem reductum rem resultant retract retractIfCan separate setPoly size sizeLess? solveLinearPolynomialEquation squareFree squareFreePart squareFreePolynomial subResultantGcd totalDegree unit? unitCanonical unitNormal univariate unmakeSUP variables vectorise zero? MODFIELD ModularField Field 0 1 * ** + − / = associates? characteristic coerce divide euclideanSize exQuo expressIdealMember exquo extendedEuclidean factor gcd inv lcm modulus multiEuclidean one? prime? principalIdeal quo recip reduce rem sizeLess? squareFree squareFreePart unit? unitCanonical unitNormal zero? MODRING ModularRing Ring 0 1 * ** + − = characteristic coerce exQuo inv modulus one? recip reduce zero? MOEBIUS MoebiusTransform Group 1 * ** / = coerce commutator conjugate eval inv moebius one? recip scale shift MRING MonoidRing Algebra CharacteristicNonZero CharacteristicZero Finite RetractableTo Ring 0 1 * ** + − = characteristic charthRoot coefficient coefficients coerce index leadingCoefficient leadingMonomial lookup map monomial monomial? monomials numberOfMonomials one? random recip reductum retract retractIfCan size terms zero? MSET Multiset MultiSetAggregate # < = any? bag brace coerce construct convert copy count dictionary difference duplicates empty empty? eq? every? extract! find insert! inspect intersect less? map map! member? members more? multiset parts reduce remove remove! removeDuplicates removeDuplicates! select select! size? subset? symmetricDifference union MPOLY MultivariatePolynomial PolynomialCategory 0 1 * ** + − / < = D associates? characteristic charthRoot coefficient coefficients coerce conditionP content convert degree differentiate discriminant eval exquo factor factorPolynomial factorSquareFreePolynomial gcd gcdPolynomial ground ground? isExpt isPlus isTimes lcm leadingCoefficient leadingMonomial mainVariable map mapExponents max min minimumDegree monicDivide monomial monomial? monomials multivariate numberOfMonomials one? prime? primitiveMonomials primitivePart recip reducedSystem reductum resultant retract retractIfCan solveLinearPolynomialEquation squareFree squareFreePart squareFreePolynomial totalDegree unit? unitCanonical unitNormal univariate variables zero? NDP NewDirectProduct DirectProductCategory 0 1 # * ** + − / < = D abs any? characteristic coerce copy count differentiate dimension di-

rectProduct dot elt empty empty? entries entry? eq? every? fill!
first index index? indices less? lookup map map! max maxIndex
member? members min minIndex more? negative? one? parts
positive? qelt qsetelt! random recip reducedSystem retract retrac-
tIfCan setelt sign size size? sup swap! unitVector zero? NDMP
NewDistributedMultivariatePolynomial PolynomialCategory 0 1 ∗ ∗∗
+ − / < = D associates? characteristic charthRoot coefficient coeffi-
cients coerce conditionP const content convert degree differentiate
discriminant eval exquo factor factorPolynomial factorSquareFreeP-
olynomial gcd gcdPolynomial ground ground? isExpt isPlus isTimes
lcm leadingCoefficient leadingMonomial mainVariable map mapEx-
ponents max min minimumDegree monicDivide monomial monomial?
monomials multivariate numberOfMonomials one? prime? primitive-
Monomials primitivePart recip reducedSystem reductum reorder re-
sultant retract retractIfCan solveLinearPolynomialEquation square-
Free squareFreePart squareFreePolynomial totalDegree unit? unit-
Canonical unitNormal univariate variables zero? NONE None Set-
Category = coerce NNI NonNegativeInteger Monoid OrderedAbelian-
MonoidSup 0 1 ∗ ∗∗ + − < = coerce divide exquo gcd max min one?
quo recip rem sup zero? OCT Octonion FullyRetractableTo Octo-
nionCategory 0 1 ∗ ∗∗ + − < = abs characteristic charthRoot coerce
conjugate convert elt eval imagE imagI imagJ imagK imagi imagj
imagk index inv lookup map max min norm octon one? random ratio-
nal rational? rationalIfCan real recip retract retractIfCan size zero?
ARRAY1 OneDimensionalArray OneDimensionalArrayAggregate # <
= any? coerce concat construct convert copy copyInto! count delete
elt empty empty? entries entry? eq? every? fill! find first index?
indices insert less? map map! max maxIndex member? members
merge min minIndex more? new oneDimensionalArray parts position
qelt qsetelt! reduce remove removeDuplicates reverse reverse! select
setelt size? sort sort! sorted? swap! ONECOMP OnePointCom-
pletion AbelianGroup FullyRetractableTo OrderedRing SetCategory
0 1 ∗ ∗∗ + − < = abs characteristic coerce finite? infinite? infinity
max min negative? one? positive? rational rational? rationalIfCan
recip retract retractIfCan sign zero? OP Operator Algebra Charac-
teristicNonZero CharacteristicZero Eltable RetractableTo Ring 0 1 ∗
∗∗ + − = characteristic charthRoot coerce elt evaluate one? opeval
recip retract retractIfCan zero? OMLO OppositeMonogenicLinear-
Operator DifferentialRing MonogenicLinearOperator 0 1 ∗ ∗∗ + − =
D characteristic coefficient coerce degree differentiate leadingCoeffi-
cient minimumDegree monomial one? op po recip reductum zero?
ORDCOMP OrderedCompletion AbelianGroup FullyRetractableTo
OrderedRing SetCategory 0 1 ∗ ∗∗ + − < = abs characteristic co-
erce finite? infinite? max min minusInfinity negative? one? plus-
Infinity positive? rational rational? rationalIfCan recip retract re-
tractIfCan sign whatInfinity zero? ODP OrderedDirectProduct Di-

rectProductCategory 0 1 # * ** + – / < = D abs any? characteristic coerce copy count differentiate dimension directProduct dot elt empty empty? entries entry? eq? every? fill! first index index? indices less? lookup map map! max maxIndex member? members min minIndex more? negative? one? parts positive? qelt qsetelt! random recip reducedSystem retract retractIfCan setelt sign size size? sup swap! unitVector zero? OVAR OrderedVariableList ConvertibleTo OrderedFinite < = coerce convert index lookup max min random size variable ODPOL OrderlyDifferentialPolynomial DifferentialPolynomialCategory RetractableTo 0 1 * ** + – / < = D associates? characteristic charthRoot coefficient coefficients coerce conditionP content degree differentialVariables differentiate discriminant eval exquo factor factorPolynomial factorSquareFreePolynomial gcd gcdPolynomial ground ground? initial isExpt isPlus isTimes isobaric? lcm leader leadingCoefficient leadingMonomial mainVariable makeVariable map mapExponents max min minimumDegree monicDivide monomial monomial? monomials multivariate numberOfMonomials one? order prime? primitiveMonomials primitivePart recip reducedSystem reductum resultant retract retractIfCan separant solveLinearPolynomialEquation squareFree squareFreePart squareFreePolynomial totalDegree unit? unitCanonical unitNormal univariate variables weight weights zero? ODVAR OrderlyDifferentialVariable DifferentialVariableCategory < = D coerce differentiate makeVariable max min order retract retractIfCan variable weight ODR OrdinaryDifferentialRing Algebra DifferentialRing Field 0 1 * ** + – / = D associates? characteristic coerce differentiate divide euclideanSize expressIdealMember exquo extendedEuclidean factor gcd inv lcm multiEuclidean one? prime? principalIdeal quo recip rem sizeLess? squareFree squareFreePart unit? unitCanonical unitNormal zero? OSI OrdSetInts OrderedSet < = coerce max min value OUTFORM OutputForm SetCategory * ** + – / < <= = > >= D SEGMENT ^= and assign blankSeparate box brace bracket center coerce commaSeparate differentiate div dot elt empty exquo hconcat height hspace infix infix? int label left matrix message messagePrint not or outputForm over overbar paren pile postfix prefix presub presuper prime print prod quo quote rarrow rem right root rspace scripts semicolonSeparate slash string sub subHeight sum super superHeight supersub vconcat vspace width zag PADIC PAdicInteger PAdicIntegerCategory 0 1 * ** + – = approximate associates? characteristic coerce complete digits divide euclideanSize expressIdealMember exquo extend extendedEuclidean gcd lcm moduloP modulus multiEuclidean one? order principalIdeal quo quotientByP recip rem sizeLess? sqrt unit? unitCanonical unitNormal zero? PADICRC PAdicRationalConstructor QuotientFieldCategory 0 1 * ** + – / < = D abs approximate associates? ceiling characteristic charthRoot coerce conditionP continuedFraction convert denom denominator differentiate divide

elt euclideanSize eval expressIdealMember exquo extendedEuclidean
factor factorPolynomial factorSquareFreePolynomial floor fraction-
Part gcd gcdPolynomial init inv lcm map max min multiEuclidean
negative? nextItem numer numerator one? patternMatch posi-
tive? prime? principalIdeal quo random recip reducedSystem rem
removeZeroes retract retractIfCan sign sizeLess? solveLinearPolyno-
mialEquation squareFree squareFreePart squareFreePolynomial unit?
unitCanonical unitNormal wholePart zero? PADICRAT PAdicRa-
tional QuotientFieldCategory 0 1 ∗ ∗∗ + − / = D approximate asso-
ciates? characteristic coerce continuedFraction denom denominator
differentiate divide euclideanSize expressIdealMember exquo extend-
edEuclidean factor fractionPart gcd inv lcm map multiEuclidean nu-
mer numerator one? prime? principalIdeal quo recip reducedSys-
tem rem removeZeroes retract retractIfCan sizeLess? squareFree
squareFreePart unit? unitCanonical unitNormal wholePart zero?
PALETTE Palette SetCategory = bright coerce dark dim hue light
pastel shade PARPCURV ParametricPlaneCurve coordinate curve
PARSCURV ParametricSpaceCurve coordinate curve PARSURF
ParametricSurface coordinate surface PFR PartialFraction Algebra
Field 0 1 ∗ ∗∗ + − / = associates? characteristic coerce compactFraction
divide euclideanSize expressIdealMember exquo extendedEuclidean
factor firstDenom firstNumer gcd inv lcm multiEuclidean nthFrac-
tionalTerm numberOfFractionalTerms one? padicFraction padically-
Expand partialFraction prime? principalIdeal quo recip rem size-
Less? squareFree squareFreePart unit? unitCanonical unitNormal
wholePart zero? PRTITION Partition ConvertibleTo OrderedCan-
cellationAbelianMonoid 0 ∗ + − < = coerce conjugate convert max min
partition pdct powers zero? PATLRES PatternMatchListResult Set-
Category = atoms coerce failed failed? lists makeResult new PATRES
PatternMatchResult SetCategory = addMatch addMatchRestricted
coerce construct destruct failed failed? getMatch insertMatch new
satisfy? union PATTERN Pattern RetractableTo SetCategory 0 1 ∗ ∗∗
+ / = addBadValue coerce constant? convert copy depth elt generic?
getBadValues hasPredicate? hasTopPredicate? inR? isExpt isList
isOp isPlus isPower isQuotient isTimes multiple? optional? optpair
patternVariable predicates quoted? resetBadValues retract retract-
IfCan setPredicates setTopPredicate symbol? topPredicate variables
withPredicates PENDTREE PendantTree BinaryRecursiveAggregate
= any? children coerce copy count cyclic? elt empty empty? eq?
every? leaf? leaves left less? map map! member? members more?
node? nodes parts ptree right setchildren! setelt setleft! setright!
setvalue! size? value PERMGRP PermutationGroup SetCategory <
<= = base coerce degree elt generators initializeGroupForWordProb-
lem member? movedPoints orbit orbits order permutationGroup
random strongGenerators wordInGenerators wordInStrongGenera-
tors wordsForStrongGenerators PERM Permutation Permutation-

Category 1 ∗ ∗∗ / < = coerce coerceImages coerceListOfPairs coercePreimagesImages commutator conjugate cycle cyclePartition cycles degree elt eval even? fixedPoints inv listRepresentation max min movedPoints numberOfCycles odd? one? orbit order recip sign sort HACKPI Pi CharacteristicZero CoercibleTo ConvertibleTo Field RealConstant RetractableTo 0 1 ∗ ∗∗ + − / = associates? characteristic coerce convert divide euclideanSize expressIdealMember exquo extendedEuclidean factor gcd inv lcm multiEuclidean one? pi prime? principalIdeal quo recip rem retract retractIfCan sizeLess? squareFree squareFreePart unit? unitCanonical unitNormal zero? ACPLOT PlaneAlgebraicCurvePlot PlottablePlaneCurveCategory coerce listBranches makeSketch refine xRange yRange PLOT3D Plot3D PlottableSpaceCurveCategory adaptive3D? coerce debug3D listBranches maxPoints3D minPoints3D numFunEvals3D plot pointPlot refine screenResolution3D setAdaptive3D setMaxPoints3D setMinPoints3D setScreenResolution3D tRange tValues xRange yRange zRange zoom PLOT Plot PlottablePlaneCurveCategory adaptive? coerce debug listBranches maxPoints minPoints numFunEvals parametric? plot plotPolar pointPlot refine screenResolution setAdaptive setMaxPoints setMinPoints setScreenResolution tRange xRange yRange zoom POINT Point PointCategory # ∗ + − < = any? coerce concat construct convert copy copyInto! count cross delete dimension dot elt empty empty? entries entry? eq? every? extend fill! find first index? indices insert length less? map map! max maxIndex member? members merge min minIndex more? new parts point position qelt qsetelt! reduce remove removeDuplicates reverse reverse! select setelt size? sort sort! sorted? swap! zero IDEAL PolynomialIdeals SetCategory ∗ ∗∗ + = backOldPos coerce contract dimension element? generalPosition generators groebner groebner? groebnerIdeal ideal in? inRadical? intersect leadingIdeal quotient relationsIdeal saturate zeroDim? PR PolynomialRing FiniteAbelianMonoidRing 0 1 ∗ ∗∗ + − / = associates? characteristic charthRoot coefficient coefficients coerce content degree exquo ground ground? leadingCoefficient leadingMonomial map mapExponents minimumDegree monomial monomial? numberOfMonomials one? primitivePart recip reductum retract retractIfCan unit? unitCanonical unitNormal zero? POLY Polynomial PolynomialCategory 0 1 ∗ ∗∗ + − / < = D associates? characteristic charthRoot coefficient coefficients coerce conditionP content convert degree differentiate discriminant eval exquo factor factorPolynomial factorSquareFreePolynomial gcd gcdPolynomial ground ground? integrate isExpt isPlus isTimes lcm leadingCoefficient leadingMonomial mainVariable map mapExponents max min minimumDegree monicDivide monomial monomial? monomials multivariate numberOfMonomials one? patternMatch prime? primitiveMonomials primitivePart recip reducedSystem reductum resultant retract retractIfCan solveLinearPolynomialEquation square-

Free squareFreePart squareFreePolynomial totalDegree unit? unitCanonical unitNormal univariate variables zero? PI PositiveInteger AbelianSemiGroup Monoid OrderedSet 1 * ** + < = coerce gcd max min one? recip PF PrimeField ConvertibleTo FiniteAlgebraicExtensionField FiniteFieldCategory 0 1 * ** + - / = Frobenius algebraic? associates? basis characteristic charthRoot coerce conditionP convert coordinates createNormalElement createPrimitiveElement definingPolynomial degree dimension discreteLog divide euclideanSize expressIdealMember exquo extendedEuclidean extensionDegree factor factorsOfCyclicGroupSize gcd generator inGroundField? index init inv lcm lookup minimalPolynomial multiEuclidean nextItem norm normal? normalElement one? order prime? primeFrobenius primitive? primitiveElement principalIdeal quo random recip rem representationType represents retract retractIfCan size sizeLess? squareFree squareFreePart tableForDiscreteLogarithm trace transcendenceDegree transcendent? unit? unitCanonical unitNormal zero? PRIMARR PrimitiveArray OneDimensionalArrayAggregate # < = any? coerce concat construct convert copy copyInto! count delete elt empty empty? entries entry? eq? every? fill! find first index? indices insert less? map map! max maxIndex member? members merge min minIndex more? new parts position qelt qsetelt! reduce remove removeDuplicates reverse reverse! select setelt size? sort sort! sorted? swap! PRODUCT Product AbelianGroup AbelianMonoid CancellationAbelianMonoid Finite Group Monoid OrderedAbelianMonoidSup OrderedSet SetCategory 0 1 * ** + - / < = coerce commutator conjugate index inv lookup makeprod max min one? random recip selectfirst selectsecond size sup zero? QFORM QuadraticForm AbelianGroup 0 * + - = coerce elt matrix quadraticForm zero? QALGSET QuasiAlgebraicSet CoercibleTo SetCategory = coerce definingEquations definingInequation empty? idealSimplify quasiAlgebraicSet setStatus simplify QUAT Quaternion QuaternionCategory 0 1 * ** + - < = D abs characteristic charthRoot coerce conjugate convert differentiate elt eval imagI imagJ imagK inv map max min norm one? quatern rational rational? rationalIfCan real recip reducedSystem retract retractIfCan zero? QEQUAT QueryEquation equation value variable QUEUE Queue QueueAggregate # = any? back bag coerce copy count dequeue! empty empty? enqueue! eq? every? extract! front insert! inspect length less? map map! member? members more? parts queue rotate! size? RADFF RadicalFunctionField FunctionFieldCategory 0 1 * ** + - / = D absolutelyIrreducible? associates? basis branchPoint? branchPointAtInfinity? characteristic characteristicPolynomial charthRoot coerce complementaryBasis convert coordinates definingPolynomial derivationCoordinates differentiate discriminant divide elt euclideanSize expressIdealMember exquo extendedEuclidean factor gcd generator genus integral? integralAtIn-

finity? integralBasis integralBasisAtInfinity integralCoordinates integralDerivationMatrix integralMatrix integralMatrixAtInfinity integralRepresents inv inverseIntegralMatrix inverseIntegralMatrixAtInfinity lcm lift minimalPolynomial multiEuclidean nonSingularModel norm normalizeAtInfinity numberOfComponents one? prime? primitivePart principalIdeal quo ramified? ramifiedAtInfinity? rank rationalPoint? rationalPoints recip reduce reduceBasisAtInfinity reducedSystem regularRepresentation rem represents retract retractIfCan singular? singularAtInfinity? sizeLess? squareFree squareFreePart trace traceMatrix unit? unitCanonical unitNormal yCoordinates zero? RADIX RadixExpansion QuotientFieldCategory 0 1 * ** + - / < = D abs associates? ceiling characteristic coerce convert cycleRagits denom denominator differentiate divide euclideanSize expressIdealMember exquo extendedEuclidean factor floor fractRadix fractRagits fractionPart gcd init inv lcm map max min multiEuclidean negative? nextItem numer numerator one? patternMatch positive? prefixRagits prime? principalIdeal quo random recip reducedSystem rem retract retractIfCan sign sizeLess? squareFree squareFreePart unit? unitCanonical unitNormal wholePart wholeRadix wholeRagits zero? RMATRIX RectangularMatrix CoercibleTo RectangularMatrixCategory VectorSpace 0 # * + - / = antisymmetric? any? coerce column copy count diagonal? dimension elt empty empty? eq? every? exquo less? listOfLists map map! matrix maxColIndex maxRowIndex member? members minColIndex minRowIndex more? ncols nrows nullSpace nullity parts qelt rank rectangularMatrix row rowEchelon size? square? symmetric? zero? REF Reference Object SetCategory = coerce deref elt ref setelt setref RULE RewriteRule Eltable RetractableTo SetCategory = coerce elt lhs pattern quotedOperators retract retractIfCan rhs rule suchThat ROMAN RomanNumeral IntegerNumberSystem 0 1 * ** + - < = D abs addmod associates? base binomial bit? characteristic coerce convert copy dec differentiate divide euclideanSize even? expressIdealMember exquo extendedEuclidean factor factorial gcd hash inc init invmod lcm length mask max min mulmod multiEuclidean negative? nextItem odd? one? patternMatch permutation positive? positiveRemainder powmod prime? principalIdeal quo random rational rational? rationalIfCan recip reducedSystem rem retract retractIfCan roman shift sign sizeLess? squareFree squareFreePart submod symmetricRemainder unit? unitCanonical unitNormal zero? RULECOLD RuleCalled SetCategory = coerce name RULESET Ruleset Eltable SetCategory = coerce elt rules ruleset FORMULA1 ScriptFormulaFormat1 Object coerce FORMULA ScriptFormulaFormat SetCategory = coerce convert display epilogue formula new prologue setEpilogue! setFormula! setPrologue! SEGBIND SegmentBinding SetCategory = coerce equation segment variable SEG Segment SegmentCategory SegmentExpansionCategory = BY SEGMENT coerce convert expand

hi high incr lo low map segment SCFRAC SemiCancelledFraction ConvertibleTo QuotientFieldCategory 0 1 ∗ ∗∗ + − / < = D abs associates? ceiling characteristic charthRoot coerce conditionP convert denom denominator differentiate divide elt euclideanSize eval expressIdealMember exquo extendedEuclidean factor factorPolynomial factorSquareFreePolynomial floor fractionPart gcd gcdPolynomial init inv lcm map max min multiEuclidean negative? nextItem normalize numer numerator one? patternMatch positive? prime? principalIdeal quo random recip reducedSystem rem retract retractIfCan sign sizeLess? solveLinearPolynomialEquation squareFree squareFreePart squareFreePolynomial unit? unitCanonical unitNormal wholePart zero? SDPOL SequentialDifferentialPolynomial DifferentialPolynomialCategory RetractableTo 0 1 ∗ ∗∗ + − / < = D associates? characteristic charthRoot coefficient coefficients coerce conditionP content degree differentialVariables differentiate discriminant eval exquo factor factorPolynomial factorSquareFreePolynomial gcd gcdPolynomial ground ground? initial isExpt isPlus isTimes isobaric? lcm leader leadingCoefficient leadingMonomial mainVariable make-Variable map mapExponents max min minimumDegree monicDivide monomial monomial? monomials multivariate numberOfMonomials one? order prime? primitiveMonomials primitivePart recip reducedSystem reductum resultant retract retractIfCan separant solveLinearPolynomialEquation squareFree squareFreePart squareFreePolynomial totalDegree unit? unitCanonical unitNormal univariate variables weight weights zero? SDVAR SequentialDifferentialVariable DifferentialVariableCategory < = D coerce differentiate make-Variable max min order retract retractIfCan variable weight SET Set FiniteSetAggregate # < = any? bag brace cardinality coerce complement construct convert copy count dictionary difference empty empty? eq? every? extract! find index insert! inspect intersect less? lookup map map! max member? members min more? parts random reduce remove remove! removeDuplicates select select! size size? subset? symmetricDifference union universe SEXOF SExpressionOf SExpressionCategory # = atom? car cdr coerce convert destruct elt eq expr float float? integer integer? list? null? pair? string string? symbol symbol? uequal SEX SExpression SExpressionCategory # = atom? car cdr coerce convert destruct elt eq expr float float? integer integer? list? null? pair? string string? symbol symbol? uequal SAE SimpleAlgebraicExtension MonogenicAlgebra 0 1 ∗ ∗∗ + − / = D associates? basis characteristic characteristicPolynomial charthRoot coerce conditionP convert coordinates createPrimitiveElement definingPolynomial derivationCoordinates differentiate discreteLog discriminant divide euclideanSize expressIdealMember exquo extendedEuclidean factor factorsOfCyclicGroupSize gcd generator index init inv lcm lift lookup minimalPolynomial multiEuclidean nextItem norm one? order prime? primeFrobenius primi-

tive? primitiveElement principalIdeal quo random rank recip reduce reducedSystem regularRepresentation rem representationType represents retract retractIfCan size sizeLess? squareFree squareFreePart tableForDiscreteLogarithm trace traceMatrix unit? unitCanonical unitNormal zero? SAOS SingletonAsOrderedSet OrderedSet < = coerce create max min SINT SingleInteger IntegerNumberSystem 0 1 * ** + − < = And D Not Or ˆ abs addmod and associates? base binomial bit? characteristic coerce convert copy dec differentiate divide euclideanSize even? expressIdealMember exquo extendedEuclidean factor factorial gcd hash inc init invmod lcm length mask max min mulmod multiEuclidean negative? nextItem not odd? one? or patternMatch permutation positive? positiveRemainder powmod prime? principalIdeal quo random rational rational? rationalIfCan recip reducedSystem rem retract retractIfCan shift sign sizeLess? squareFree squareFreePart submod symmetricRemainder unit? unitCanonical unitNormal xor zero? SMP SparseMultivariatePolynomial PolynomialCategory 0 1 * ** + − / < = D associates? characteristic charthRoot coefficient coefficients coerce conditionP content convert degree differentiate discriminant eval exquo factor factorPolynomial factorSquareFreePolynomial gcd gcdPolynomial ground ground? isExpt isPlus isTimes lcm leadingCoefficient leadingMonomial mainVariable map mapExponents max min minimumDegree monicDivide monomial monomial? monomials multivariate numberOfMonomials one? patternMatch prime? primitiveMonomials primitivePart recip reducedSystem reductum resultant retract retractIfCan solveLinearPolynomialEquation squareFree squareFreePart squareFreePolynomial totalDegree unit? unitCanonical unitNormal univariate variables zero? SMTS SparseMultivariateTaylorSeries MultivariateTaylorSeriesCategory 0 1 * ** + − / = D acos acosh acot acoth acsc acsch asec asech asin asinh associates? atan atanh characteristic charthRoot coefficient coerce complete cos cosh cot coth csc csch csubst degree differentiate eval exp exquo extend fintegrate integrate leadingCoefficient leadingMonomial log map monomial monomial? nthRoot one? order pi pole? polynomial recip reductum sec sech sin sinh sqrt tan tanh unit? unitCanonical unitNormal variables zero? STBL SparseTable TableAggregate # = any? bag coerce construct copy count dictionary elt empty empty? entries entry? eq? every? extract! fill! find first index? indices insert! inspect key? keys less? map map! maxIndex member? members minIndex more? parts qelt qsetelt! reduce remove remove! removeDuplicates search select select! setelt size? swap! table SUP SparseUnivariatePolynomial UnivariatePolynomialCategory 0 1 * ** + − / < = D associates? characteristic charthRoot coefficient coefficients coerce composite conditionP content degree differentiate discriminant divide divideExponents elt euclideanSize eval expressIdealMember exquo extendedEuclidean factor factorPolynomial factorSquareFreePolynomial gcd gcdPolynomial ground

ground? init integrate isExpt isPlus isTimes lcm leadingCoefficient leadingMonomial mainVariable makeSUP map mapExponents max min minimumDegree monicDivide monomial monomial? monomials multiEuclidean multiplyExponents multivariate nextItem numberOfMonomials one? order outputForm prime? primitiveMonomials primitivePart principalIdeal pseudoDivide pseudoQuotient pseudoRemainder quo recip reducedSystem reductum rem resultant retract retractIfCan separate sizeLess? solveLinearPolynomialEquation squareFree squareFreePart squareFreePolynomial subResultantGcd totalDegree unit? unitCanonical unitNormal univariate unmakeSUP variables vectorise zero? SUTS SparseUnivariateTaylorSeries UnivariateTaylorSeriesCategory 0 1 * ** + − / = D acos acosh acot acoth acsc acsch approximate asec asech asin asinh associates? atan atanh center characteristic charthRoot coefficient coefficients coerce complete cos cosh cot coth csc csch degree differentiate elt eval exp exquo extend integrate leadingCoefficient leadingMonomial log map monomial monomial? multiplyCoefficients multiplyExponents nthRoot one? order pi pole? polynomial quoByVar recip reductum sec sech series sin sinh sqrt tan tanh terms truncate unit? unitCanonical unitNormal variable variables zero? SQMATRIX SquareMatrix CoercibleTo SquareMatrixCategory 0 1 # * ** + − / = D antisymmetric? any? characteristic coerce column copy count determinant diagonal diagonal? diagonalMatrix diagonalProduct differentiate elt empty empty? eq? every? exquo inverse less? listOfLists map map! matrix maxColIndex maxRowIndex member? members minColIndex minRowIndex minordet more? ncols nrows nullSpace nullity one? parts qelt rank recip reducedSystem retract retractIfCan row rowEchelon scalarMatrix size? square? squareMatrix symmetric? trace transpose zero? STACK Stack StackAggregate # = any? bag coerce copy count depth empty empty? eq? every? extract! insert! inspect less? map map! member? members more? parts pop! push! size? stack top STREAM Stream LazyStreamAggregate # = any? child? children coerce complete concat concat! cons construct convert copy count cycleEntry cycleLength cycleSplit! cycleTail cyclic? delay delete distance elt empty empty? entries entry? eq? every? explicitEntries? explicitlyEmpty? explicitlyFinite? extend fill! filterUntil filterWhile find findCycle first frst generate index? indices insert last lazy? lazyEvaluate leaf? less? map map! maxIndex member? members minIndex more? new node? nodes numberOfComputedEntries output parts possiblyInfinite? qelt qsetelt! reduce remove removeDuplicates repeating repeating? rest rst second select setchildren! setelt setfirst! setlast! setrest! setvalue! showAll? showAllElements size? split! swap! tail third value STRTBL StringTable TableAggregate # = any? bag coerce construct copy count dictionary elt empty empty? entries entry? eq? every? extract! fill! find first index? indices insert! inspect key? keys less? map map! maxIndex member?

members minIndex more? parts qelt qsetelt! reduce remove remove! removeDuplicates search select select! setelt size? swap! table STRING String StringCategory # < = any? coerce concat construct copy copyInto! count delete elt empty empty? entries entry? eq? every? fill! find first index? indices insert leftTrim less? lowerCase lowerCase! map map! match? max maxIndex member? members merge min minIndex more? new parts position prefix? qelt qsetelt! reduce remove removeDuplicates replace reverse reverse! rightTrim select setelt size? sort sort! sorted? split string substring? suffix? swap! trim upperCase upperCase! COMPPROP SubSpaceComponentProperty SetCategory = close closed? coerce copy new solid solid? SUBSPACE SubSpace SetCategory = addPoint addPoint2 addPointLast birth child children closeComponent coerce deepCopy defineProperty extractClosed extractIndex extractPoint extractProperty internal? leaf? level merge modifyPoint new numberOfChildren parent pointData root? separate shallowCopy subspace traverse SUCH SuchThat SetCategory = coerce construct lhs rhs SYMBOL Symbol ConvertibleTo OrderedSet PatternMatchable < = argscript coerce convert elt list max min name new patternMatch resetNew script scripted? scripts string subscript superscript SYMPOLY SymmetricPolynomial FiniteAbelianMonoidRing 0 1 * ** + − / = associates? characteristic charthRoot coefficient coefficients coerce content degree exquo ground ground? leadingCoefficient leadingMonomial map mapExponents minimumDegree monomial monomial? numberOfMonomials one? primitivePart recip reductum retract retractIfCan unit? unitCanonical unitNormal zero? TABLEAU Tableau Object coerce listOfLists tableau TABLE Table TableAggregate # = any? bag coerce construct copy count dictionary elt empty empty? entries entry? eq? every? extract! fill! find first index? indices insert! inspect key? keys less? map map! maxIndex member? members minIndex more? parts qelt qsetelt! reduce remove remove! removeDuplicates search select select! setelt size? swap! table TS TaylorSeries MultivariateTaylorSeriesCategory 0 1 * ** + − / = D acos acosh acot acoth acsc acsch asec asech asin asinh associates? atan atanh characteristic charthRoot coefficient coerce complete cos cosh cot coth csc csch degree differentiate eval exp exquo extend fintegrate integrate leadingCoefficient leadingMonomial log map monomial monomial? nthRoot one? order pi pole? polynomial recip reductum sec sech sin sinh sqrt tan tanh unit? unitCanonical unitNormal variables zero? TEX1 TexFormat1 Object coerce TEX TexFormat SetCategory = coerce convert display epilogue new prologue setEpilogue! setPrologue! setTex! tex TEXTFILE TextFile FileCategory = close! coerce endOfFile? iomode name open read! readIfCan! readLine! readLineIfCan! reopen! write! writeLine! VIEW3D ThreeDimensionalViewport SetCategory = axes clipSurface close coerce colorDef controlPanel diagonals dimensions drawStyle eyeDistance hitherPlane

intensity key lighting makeViewport3D modifyPointData move options outlineRender perspective reset resize rotate showClipRegion showRegion subspace title translate viewDeltaXDefault viewDeltaY-Default viewPhiDefault viewThetaDefault viewZoomDefault viewpoint viewport3D write zoom SPACE3 ThreeSpace ThreeSpaceCategory = check closedCurve closedCurve? coerce components composite composites copy create3Space curve curve? enterPointData lllip lllp llprop lp lprop merge mesh mesh? modifyPointData numberOfComponents numberOfComposites objects point point? polygon polygon? subspace TREE Tree RecursiveAggregate # = any? children coerce copy count cyclic? elt empty empty? eq? every? leaf? leaves less? map map! member? members more? node? nodes parts setchildren! setelt setvalue! size? tree value TUBE TubePlot closed? getCurve listLoops open? setClosed tube TUPLE Tuple CoercibleTo SetCategory = coerce length select ARRAY2 TwoDimensionalArray TwoDimensionalArrayCategory # = any? coerce column copy count elt empty empty? eq? every? fill! less? map map! maxColIndex maxRowIndex member? members minColIndex minRowIndex more? ncols new nrows parts qelt qsetelt! row setColumn! setRow! setelt size? VIEW2D TwoDimensionalViewport SetCategory = axes close coerce connect controlPanel dimensions getGraph graphState graphStates graphs key makeViewport2D move options points putGraph region reset resize scale show title translate units viewport2D write ULSCONS UnivariateLaurentSeriesConstructor UnivariateLaurentSeriesConstructorCategory 0 1 * ** + - / < = D abs acos acosh acot acoth acsc acsch approximate asec asech asin asinh associates? atan atanh ceiling center characteristic charthRoot coefficient coerce complete conditionP convert cos cosh cot coth csc csch degree denom denominator differentiate divide elt euclideanSize eval exp expressIdealMember exquo extend extendedEuclidean factor factorPolynomial factorSquareFreePolynomial floor fractionPart gcd gcdPolynomial init integrate inv laurent lcm leadingCoefficient leadingMonomial log map max min monomial monomial? multiEuclidean multiplyCoefficients multiplyExponents negative? nextItem nthRoot numer numerator one? order patternMatch pi pole? positive? prime? principalIdeal quo random rationalFunction recip reducedSystem reductum rem removeZeroes retract retractIfCan sec sech series sign sin sinh sizeLess? solveLinearPolynomialEquation sqrt squareFree squareFreePart squareFreePolynomial tan tanh taylor taylorIfCan taylorRep terms truncate unit? unitCanonical unitNormal variable variables wholePart zero? ULS UnivariateLaurentSeries UnivariateLaurentSeriesConstructorCategory 0 1 * ** + - / = D acos acosh acot acoth acsc acsch approximate asec asech asin asinh associates? atan atanh center characteristic charthRoot coefficient coerce complete cos cosh cot coth csc csch degree denom denominator differentiate divide elt euclidean-

Size eval exp expressIdealMember exquo extend extendedEuclidean factor gcd integrate inv laurent lcm leadingCoefficient leadingMonomial log map monomial monomial? multiEuclidean multiplyCoefficients multiplyExponents nthRoot numer numerator one? order pi pole? prime? principalIdeal quo rationalFunction recip reducedSystem reductum rem removeZeroes retract retractIfCan sec sech series sin sinh sizeLess? sqrt squareFree squareFreePart tan tanh taylor taylorIfCan taylorRep terms truncate unit? unitCanonical unitNormal variable variables zero? UP UnivariatePolynomial UnivariatePolynomialCategory 0 1 ∗ ∗∗ + − / < = D associates? characteristic charthRoot coefficient coefficients coerce composite conditionP content degree differentiate discriminant divide divideExponents elt euclideanSize eval expressIdealMember exquo extendedEuclidean factor factorPolynomial factorSquareFreePolynomial gcd gcdPolynomial ground ground? init integrate isExpt isPlus isTimes lcm leadingCoefficient leadingMonomial mainVariable makeSUP map mapExponents max min minimumDegree monicDivide monomial monomial? monomials multiEuclidean multiplyExponents multivariate nextItem numberOfMonomials one? order prime? primitiveMonomials primitivePart principalIdeal pseudoDivide pseudoQuotient pseudoRemainder quo recip reducedSystem reductum rem resultant retract retractIfCan separate sizeLess? solveLinearPolynomialEquation squareFree squareFreePart squareFreePolynomial subResultantGcd totalDegree unit? unitCanonical unitNormal univariate unmakeSUP variables vectorise zero? UPXSCONS UnivariatePuiseuxSeriesConstructor UnivariatePuiseuxSeriesConstructorCategory 0 1 ∗ ∗∗ + − / = D acos acosh acot acoth acsc acsch approximate asec asech asin asinh associates? atan atanh center characteristic charthRoot coefficient coerce complete cos cosh cot coth csc csch degree differentiate divide elt euclideanSize eval exp expressIdealMember exquo extend extendedEuclidean factor gcd integrate inv laurent laurentIfCan laurentRep lcm leadingCoefficient leadingMonomial log map monomial monomial? multiEuclidean multiplyExponents nthRoot one? order pi pole? prime? principalIdeal puiseux quo rationalPower recip reductum rem retract retractIfCan sec sech series sin sinh sizeLess? sqrt squareFree squareFreePart tan tanh terms truncate unit? unitCanonical unitNormal variable variables zero? UPXS UnivariatePuiseuxSeries UnivariatePuiseuxSeriesConstructorCategory 0 1 ∗ ∗∗ + − / = D acos acosh acot acoth acsc acsch approximate asec asech asin asinh associates? atan atanh center characteristic charthRoot coefficient coerce complete cos cosh cot coth csc csch degree differentiate divide elt euclideanSize eval exp expressIdealMember exquo extend extendedEuclidean factor gcd integrate inv laurent laurentIfCan laurentRep lcm leadingCoefficient leadingMonomial log map monomial monomial? multiEuclidean multiplyExponents nthRoot one? order pi pole? prime? principalIdeal puiseux quo rationalPower recip re-

ductum rem retract retractIfCan sec sech series sin sinh sizeLess? sqrt squareFree squareFreePart tan tanh terms truncate unit? unit-Canonical unitNormal variable variables zero? UTS UnivariateTaylorSeries UnivariateTaylorSeriesCategory 0 1 * ** + - / = D acos acosh acot acoth acsc acsch approximate asec asech asin asinh associates? atan atanh center characteristic charthRoot coefficient coefficients coerce complete cos cosh cot coth csc csch degree differentiate elt eval evenlambert exp exquo extend generalLambert integrate invmultisect lagrange lambert leadingCoefficient leadingMonomial log map monomial monomial? multiplyCoefficients multiplyExponents multisect nthRoot oddlambert one? order pi pole? polynomial quoByVar recip reductum revert sec sech series sin sinh sqrt tan tanh terms truncate unit? unitCanonical unitNormal univariatePolynomial variable variables zero? UNISEG UniversalSegment SegmentCategory SegmentExpansionCategory = BY SEGMENT coerce convert expand hasHi hi high incr lo low map segment VARIABLE Variable CoercibleTo SetCategory = coerce variable VECTOR Vector VectorCategory # * + - < = any? coerce concat construct convert copy copyInto! count delete dot elt empty empty? entries entry? eq? every? fill! find first index? indices insert less? map map! max maxIndex member? members merge min minIndex more? new parts position qelt qsetelt! reduce remove removeDuplicates reverse reverse! select setelt size? sort sort! sorted? swap! vector zero VOID Void coerce void

Appendix D

Packages

This is a listing of all packages in the Axiom library at the time this book was produced. Use the Browse facility (described in Chapter 14 on page 943) to get more information about these constructors.

This sample entry will help you read the following table:

PackageNamePackageAbbreviation:Category$_1$... Category$_N$ *with* op$_1$... op$_M$

where

PackageName	is the full package name, e.g., PadeApproximantPackage.
PackageAbbreviation	is the package abbreviation, e.g., PADEPAC.
Category$_i$	is a category to which the package belongs.
op$_j$	is an operation exported by the package.

Packages

AF AlgebraicFunction ∗∗ belong? definingPolynomial inrootof iroot minPoly operator rootOf INTHERAL AlgebraicHermiteIntegration HermiteIntegrate INTALG AlgebraicIntegrate algintegrate palginfieldint palgintegrate INTAF AlgebraicIntegration algint ALGMANIP AlgebraicManipulations ratDenom ratPoly rootKerSimp rootSimp rootSplit ALGMFACT AlgebraicMultFact factor ALGPKG AlgebraPackage basisOfCenter basisOfCentroid basisOfCommutingElements basisOfLeftAnnihilator basisOfLeftNucleus basisOfLeftNucloid basisOfMiddleNucleus basisOfNucleus basisOfRightAnnihilator basisOfRightNucleus basisOfRightNucloid biRank doubleRank leftRank radicalOfLeftTraceForm rightRank weakBiRank ALGFACT AlgFactor doublyTransitive? factor split ANY1 AnyFunctions1 coerce retract retractIfCan retractable? APPRULE ApplyRules applyRules localUnquote PMPRED AttachPredicates suchThat BALFACT BalancedFactorisation balancedFactorisation BOP1 BasicOperatorFunctions1 constantOpIfCan constantOperator derivative evaluate BEZOUT BezoutMatrix bezoutDiscriminant bezoutMatrix bezoutRe-

sultant BOUNDZRO BoundIntegerRoots integerBound CARTEN2
CartesianTensorFunctions2 map reshape CHVAR ChangeOfVari-
able chvar eval goodPoint mkIntegral radPoly rootPoly CHARPOL
CharacteristicPolynomialPackage characteristicPolynomial CVMP
CoerceVectorMatrixPackage coerce coerceP COMBF Combinato-
rialFunction ∗∗ belong? binomial factorial factorials iibinom iid-
prod iidsum iifact iiperm iipow ipow operator permutation prod-
uct summation CDEN CommonDenominator clearDenominator com-
monDenominator splitDenominator COMMONOP CommonOper-
ators operator COMMUPC CommuteUnivariatePolynomialCate-
gory swap COMPFACT ComplexFactorization factor COMPLEX2
ComplexFunctions2 map CINTSLPE ComplexIntegerSolveLinear-
PolynomialEquation solveLinearPolynomialEquation CRFP Com-
plexRootFindingPackage complexZeros divisorCascade factor gra-
effe norm pleskenSplit reciprocalPolynomial rootRadius schwer-
punkt setErrorBound startPolynomial CMPLXRT ComplexRoot-
Package complexZeros ODECONST ConstantLODE constDsolve
COORDSYS CoordinateSystems bipolar bipolarCylindrical carte-
sian conical cylindrical elliptic ellipticCylindrical oblateSpheroidal
parabolic parabolicCylindrical paraboloidal polar prolateSpheroidal
spherical toroidal CRAPACK CRApackage chineseRemainder mod-
Tree multiEuclideanTree CYCLES CycleIndicators SFunction alter-
nating cap complete cup cyclic dihedral elementary eval graphs pow-
erSum skewSFunction wreath CSTTOOLS CyclicStreamTools com-
puteCycleEntry computeCycleLength cycleElt CYCLOTOM Cyclo-
tomicPolynomialPackage cyclotomic cyclotomicDecomposition cyclo-
tomicFactorization DEGRED DegreeReductionPackage expand re-
duce DIOSP DiophantineSolutionPackage dioSolve DIRPROD2 Di-
rectProductFunctions2 map reduce scan DLP DiscreteLogarithm-
Package shanksDiscLogAlgorithm DISPLAY DisplayPackage bright
center copies newLine say sayLength DDFACT DistinctDegreeFac-
torize distdfact exptMod factor irreducible? separateDegrees sepa-
rateFactors tracePowMod DBLRESP DoubleResultantPackage dou-
bleResultant DRAWHACK DrawNumericHack coerce DROPT0
DrawOptionFunctions0 adaptive clipBoolean coordinate curveCol-
orPalette pointColorPalette ranges space style title toScale tube-
Points tubeRadius units var1Steps var2Steps DROPT1 DrawOp-
tionFunctions1 option EP EigenPackage characteristicPolynomial
eigenvalues eigenvector eigenvectors inteigen ODEEF Elementary-
FunctionODESolver solve SIGNEF ElementaryFunctionSign sign
EFSTRUC ElementaryFunctionStructurePackage normalize realEle-
mentary rischNormalize validExponential EFUTS ElementaryFunc-
tionsUnivariateTaylorSeries ∗∗ acos acosh acot acoth acsc acsch
asec asech asin asinh atan atanh cos cosh cot coth csc csch exp
log sec sech sin sincos sinh sinhcosh tan tanh EF ElementaryFunc-
tion acos acosh acot acoth acsc acsch asec asech asin asinh atan

atanh belong? cos cosh cot coth csc csch exp iiacos iiacosh iiacot iiacoth iiacsc iiacsch iiasec iiasech iiasin iiasinh iiatan iiatanh iicos iicosh iicot iicoth iicsc iicsch iiexp iilog iisec iisech iisin iisinh iitan iitanh log operator pi sec sech sin sinh specialTrigs tan tanh IN-TEF ElementaryIntegration lfextendedint lfextlimint lfinfieldint lfintegrate lflimitedint RDEEF ElementaryRischDE rischDE ELFUTS EllipticFunctionsUnivariateTaylorSeries cn dn sn sncndn EQ2 EquationFunctions2 map ERROR ErrorFunctions error GBEUCLID EuclideanGroebnerBasisPackage euclideanGroebner euclideanNormalForm EVALCYC EvaluateCycleIndicators eval EXPR2 ExpressionFunctions2 map ES1 ExpressionSpaceFunctions1 map ES2 ExpressionSpaceFunctions2 map EXPRODE ExpressionSpaceODESolver seriesSolve EXPR2UPS ExpressionToUnivariatePowerSeries laurent puiseux series taylor EXPRTUBE ExpressionTubePlot constantToUnaryFunction tubePlot FR2 FactoredFunctions2 map FACTFUNC FactoredFunctions log nthRoot FRUTIL FactoredFunctionUtilities mergeFactors refine FACUTIL FactoringUtilities completeEval degree lowerPolynomial normalDeriv raisePolynomial ran variables FORDER FindOrderFinite order FDIV2 FiniteDivisorFunctions2 map FFF FiniteFieldFunctions createMultiplicationMatrix createMultiplicationTable createZechTable sizeMultiplication FFHOM FiniteFieldHomomorphisms coerce FFPOLY2 FiniteFieldPolynomialPackage2 rootOfIrreduciblePoly FFPOLY FiniteFieldPolynomialPackage createIrreduciblePoly createNormalPoly createNormalPrimitivePoly createPrimitiveNormalPoly createPrimitivePoly leastAffineMultiple nextIrreduciblePoly nextNormalPoly nextNormalPrimitivePoly nextPrimitiveNormalPoly nextPrimitivePoly normal? numberOfIrreduciblePoly numberOfNormalPoly numberOfPrimitivePoly primitive? random reducedQPowers FFSLPE FiniteFieldSolveLinearPolynomialEquation solveLinearPolynomialEquation FLAGG2 FiniteLinearAggregateFunctions2 map reduce scan FLASORT FiniteLinearAggregateSort heapSort quickSort shellSort FSAGG2 FiniteSetAggregateFunctions2 map reduce scan FLOATCP FloatingComplexPackage complexRoots complexSolve FLOATRP FloatingRealPackage realRoots solve FRIDEAL2 FractionalIdealFunctions2 map FRAC2 FractionFunctions2 map FSPECF FunctionalSpecialFunction Beta Gamma abs airyAi airyBi belong? besselI besselJ besselK besselY digamma iiGamma iiabs operator polygamma FFCAT2 FunctionFieldCategoryFunctions2 map FFINTBAS FunctionFieldIntegralBasis integralBasis PMASSFS FunctionSpaceAssertions assert constant multiple optional PMPREDFS FunctionSpaceAttachPredicates suchThat FSCINT FunctionSpaceComplexIntegration complexIntegrate internalIntegrate FS2 FunctionSpaceFunctions2 map FSINT FunctionSpaceIntegration integrate FSPRMELT FunctionSpacePrimitiveElement primitiveElement FSRED FunctionSpaceReduce bringDown newReduc SUMFS

FunctionSpaceSum   sum FS2UPS FunctionSpaceToUnivariatePow-
erSeries   exprToGenUPS exprToUPS FSUPFACT FunctionSpace-
UnivariatePolynomialFactor   ffactor qfactor GAUSSFAC Gaussian-
FactorizationPackage factor prime? sumSquares GHENSEL Gen-
eralHenselPackage   HenselLift completeHensel GENPGCD Gener-
alPolynomialGcdPackage gcdPolynomial randomR GENUPS Gen-
erateUnivariatePowerSeries laurent puiseux series taylor GENEEZ
GenExEuclid   compBound reduction solveid tablePow testModulus
GENUFACT GenUFactorize   factor INTG0 GenusZeroIntegration
palgLODE0 palgRDE0 palgextint0 palgint0 palglimint0 GOSPER
GosperSummationMethod   GospersMethod GRDEF GraphicsDe-
faults adaptive clipPointsDefault drawToScale maxPoints minPoints
screenResolution GRAY GrayCode firstSubsetGray nextSubsetGray
GBF GroebnerFactorizationPackage factorGroebnerBasis groebner-
Factorize GBINTERN GroebnerInternalPackage credPol critB crit-
BonD critM critMTonD1 critMonD1 critT critpOrder fprindINFO
gbasis hMonic lepol makeCrit minGbasis prinb prindINFO prin-
polINFO prinshINFO redPo redPol sPol updatD updatF virtu-
alDegree GB GroebnerPackage groebner normalForm GROEBSOL
GroebnerSolve genericPosition groebSolve testDim HB HallBasis
generate inHallBasis? lfunc HEUGCD HeuGcd content contprim
gcd gcdcofact gcdcofactprim gcdprim lintgcd IDECOMP IdealDecom-
positionPackage primaryDecomp prime? radical zeroDimPrimary?
zeroDimPrime? INCRMAPS IncrementingMaps increment incre-
mentBy ITFUN2 InfiniteTupleFunctions2 map ITFUN3 InfiniteTu-
pleFunctions3 map INFINITY Infinity infinity minusInfinity plusIn-
finity IALGFACT InnerAlgFactor factor ICDEN InnerCommonDe-
nominator clearDenominator commonDenominator splitDenomina-
tor IMATLIN InnerMatrixLinearAlgebraFunctions determinant in-
verse nullSpace nullity rank rowEchelon IMATQF InnerMatrixQuo-
tientFieldFunctions inverse nullSpace nullity rank rowEchelon IN-
MODGCD InnerModularGcd modularGcd reduction INNMFACT
InnerMultFact factor INBFF InnerNormalBasisFieldFunctions ∗ ∗∗
/ basis dAndcExp expPot index inv lookup minimalPolynomial norm
normal? normalElement pol qPot random repSq setFieldInfo trace xn
INEP InnerNumericEigenPackage charpol innerEigenvectors INFSP
InnerNumericFloatSolvePackage   innerSolve innerSolve1 makeEq
INPSIGN InnerPolySign signAround ISUMP InnerPolySum sum
ITRIGMNP InnerTrigonometricManipulations F2FG FG2F GF2FG
explogs2trigs trigs2explogs INFORM1 InputFormFunctions1 inter-
pret packageCall COMBINAT IntegerCombinatoricFunctions bino-
mial factorial multinomial partition permutation stirling1 stirling2
INTFACT IntegerFactorizationPackage BasicMethod PollardSmall-
Factor factor squareFree ZLINDEP IntegerLinearDependence lin-
earDependenceOverZ linearlyDependentOverZ? solveLinearlyOverQ
INTHEORY IntegerNumberTheoryFunctions bernoulli chineseRe-

mainder divisors euler eulerPhi fibonacci harmonic jacobi legendre moebiusMu numberOfDivisors sumOfDivisors sumOfKthPowerDivisors PRIMES IntegerPrimesPackage nextPrime prevPrime prime? primes INTRET IntegerRetractions integer integer? integerIfCan IROOT IntegerRoots approxNthRoot approxSqrt perfectNthPower? perfectNthRoot perfectSqrt perfectSquare? IBATOOL IntegralBasisTools diagonalProduct idealiser leastPower IR2 IntegrationResultFunctions2 map IRRF2F IntegrationResultRFToFunction complexExpand complexIntegrate expand integrate split IR2F IntegrationResultToFunction complexExpand expand split INTTOOLS IntegrationTools kmax ksec mkPrim union vark varselect INVLAPLA InverseLaplaceTransform inverseLaplace IRREDFFX IrredPolyOverFiniteField generateIrredPoly IRSN IrrRepSymNatPackage dimensionOfIrreducibleRepresentation irreducibleRepresentation KERNEL2 KernelFunctions2 constantIfCan constantKernel KOVACIC Kovacic kovacic LAPLACE LaplaceTransform laplace LEADCDET LeadingCoefDetermination distFact polCase LINDEP LinearDependence linearDependence linearlyDependent? solveLinear LPEFRAC LinearPolynomialEquationByFractions solveLinearPolynomialEquationByFractions LSMP LinearSystemMatrixPackage aSolution hasSolution? rank solve LSPP LinearSystemPolynomialPackage linSolve LGROBP LinGrobnerPackage anticoord choosemon computeBasis coordinate groebgen intcompBasis linGenPos minPol totolex transform LF LiouvillianFunction Ci Ei Si belong? dilog erf integral li operator LIST2 ListFunctions2 map reduce scan LIST3 ListFunctions3 map LIST2MAP ListToMap match MKBCFUNC MakeBinaryCompiledFunction binaryFunction compiledFunction MKFLCFN MakeFloatCompiledFunction makeFloatFunction MKFUNC MakeFunction function MKRECORD MakeRecord makeRecord MKUCFUNC MakeUnaryCompiledFunction compiledFunction unaryFunction MAPPKG1 MappingPackage1 ** coerce fixedPoint id nullary recur MAPPKG2 MappingPackage2 const constant curry diag MAPPKG3 MappingPackage3 * constantLeft constantRight curryLeft curryRight twist MAPHACK1 MappingPackageInternalHacks1 iter recur MAPHACK2 MappingPackageInternalHacks2 arg1 arg2 MAPHACK3 MappingPackageInternalHacks3 comp MATCAT2 MatrixCategoryFunctions2 map reduce MCDEN MatrixCommonDenominator clearDenominator commonDenominator splitDenominator MATLIN MatrixLinearAlgebraFunctions determinant inverse minordet nullSpace nullity rank rowEchelon MTHING MergeThing mergeDifference MESH MeshCreationRoutinesForThreeDimensions meshFun2Var meshPar1Var meshPar2Var ptFunc MDDFACT ModularDistinctDegreeFactorizer ddFact exptMod factor gcd separateFactors MHROWRED ModularHermitianRowReduction rowEch rowEchelon MRF2 MonoidRingFunctions2 map MSYSCMD MoreSystemCommands systemCommand MPC2

MPolyCatFunctions2 map reshape **MPC3 MPolyCatFunctions3**
map **MPRFF MPolyCatRationalFunctionFactorizer** factor pushdown pushdterm pushucoef pushuconst pushup totalfract **MRATFAC
MRationalFactorize** factor **MFINFACT MultFiniteFactorize** factor **MMAP MultipleMap** map **MULTFACT MultivariateFactorize**
factor **MLIFT MultivariateLifting** corrPoly lifting lifting1 **MULT-
SQFR MultivariateSquareFree** squareFree squareFreePrim **NCO-
DIV NonCommutativeOperatorDivision** leftDivide leftExactQuotient leftGcd leftLcm leftQuotient leftRemainder **NONE1 NoneFunctions1** coerce **NODE1 NonLinearFirstOrderODESolver** solve **NLIN-
SOL NonLinearSolvePackage** solve solveInField **NPCOEF NPCoef**
listexp npcoef **NFINTBAS NumberFieldIntegralBasis** discriminant
integralBasis **NUMFMT NumberFormats** FormatArabic FormatRoman ScanArabic ScanRoman **NTPOLFN NumberTheoreticPolynomialFunctions** bernoulliB cyclotomic eulerE **NUMODE NumericalOrdinaryDifferentialEquations** rk4 rk4a rk4f rk4qc **NUMQUAD
NumericalQuadrature** aromberg asimpson atrapezoidal romberg
rombergo simpson simpsono trapezoidal trapezoidalo **NCEP NumericComplexEigenPackage** characteristicPolynomial complexEigenvalues complexEigenvectors **NCNTFRAC NumericContinuedFraction** continuedFraction **NREP NumericRealEigenPackage** characteristicPolynomial realEigenvalues realEigenvectors **NUMTUBE NumericTubePlot** tube **NUMERIC Numeric** complexNumeric numeric
OCTCT2 OctonionCategoryFunctions2 map **ODEINT ODEIntegration** expint int **ODETOOLS ODETools** particularSolution variationOfParameters wronskianMatrix **ARRAY12 OneDimensionalArrayFunctions2** map reduce scan **ONECOMP2 OnePointCompletionFunctions2** map **OPQUERY OperationsQuery** getDatabase **OR-
DCOMP2 OrderedCompletionFunctions2** map **ORDFUNS OrderingFunctions** pureLex reverseLex totalLex **ORTHPOL OrthogonalPolynomialFunctions** ChebyshevU chebyshevT hermiteH laguerreL legendreP **OUT OutputPackage** output **PADEPAC PadeApproximantPackage** pade **PADE PadeApproximants** pade padecf **YS-
TREAM ParadoxicalCombinatorsForStreams** Y **PARTPERM PartitionsAndPermutations** conjugate conjugates partitions permutations sequences shuffle shufflein **PATTERN1 PatternFunctions1** addBadValue badValues predicate satisfy? suchThat **PATTERN2 PatternFunctions2** map **PMASS PatternMatchAssertions** assert constant multiple optional **PMFS PatternMatchFunctionSpace** patternMatch **PMINS PatternMatchIntegerNumberSystem** patternMatch
PMKERNEL PatternMatchKernel patternMatch **PMLSAGG PatternMatchListAggregate** patternMatch **PMPLCAT PatternMatchPolynomialCategory** patternMatch **PMDOWN PatternMatchPushDown** fixPredicate patternMatch **PMQFCAT PatternMatchQuotientFieldCategory** patternMatch **PATRES2 PatternMatchResultFunctions2** map **PMSYM PatternMatchSymbol** patternMatch **PM-**

TOOLS PatternMatchTools patternMatch patternMatchTimes PAT-
MATCH PatternMatch Is is? PERMAN Permanent permanent
PGE PermutationGroupExamples abelianGroup alternatingGroup
cyclicGroup dihedralGroup janko2 mathieu11 mathieu12 mathieu22
mathieu23 mathieu24 rubiksGroup symmetricGroup youngGroup PI-
COERCE PiCoercions coerce PLOT1 PlotFunctions1 plot plotPo-
lar PLOTTOOL PlotTools calcRanges PTFUNC2 PointFunctions2
map PTPACK PointPackage color hue phiCoord rCoord shade
thetaCoord xCoord yCoord zCoord PFOQ PointsOfFiniteOrderRa-
tional order torsion? torsionIfCan PFOTOOLS PointsOfFinite-
OrderTools badNum doubleDisc getGoodPrime mix polyred PFO
PointsOfFiniteOrder order torsion? torsionIfCan POLTOPOL
PolToPol dmpToNdmp dmpToP ndmpToDmp ndmpToP pToDmp
pToNdmp PGROEB PolyGroebner lexGroebner totalGroebner
PAN2EXPR PolynomialAN2Expression coerce POLYLIFT Poly-
nomialCategoryLifting map POLYCATQ PolynomialCategoryQuo-
tientFunctions isExpt isPlus isPower isTimes mainVariable multivari-
ate univariate variables PFBRU PolynomialFactorizationByRecursio-
nUnivariate bivariateSLPEBR factorByRecursion factorSFBRlcU-
nit factorSquareFreeByRecursion randomR solveLinearPolynomialE-
quationByRecursion PFBR PolynomialFactorizationByRecursion bi-
variateSLPEBR factorByRecursion factorSFBRlcUnit factorSquare-
FreeByRecursion randomR solveLinearPolynomialEquationByRecur-
sion POLY2 PolynomialFunctions2 map PGCD PolynomialGcd-
Package gcd gcdPrimitive PINTERPA PolynomialInterpolation-
Algorithms LagrangeInterpolation PINTERP PolynomialInterpola-
tion interpolate PNTHEORY PolynomialNumberTheoryFunctions
bernoulli chebyshevT chebyshevU cyclotomic euler fixedDivisor
hermite laguerre legendre POLYROOT PolynomialRoots froot
qroot rroot SOLVEFOR PolynomialSolveByFormulas aCubic aLinear
aQuadratic aQuartic aSolution cubic linear mapSolve quadratic quar-
tic solve PSQFR PolynomialSquareFree squareFree POLY2UP Poly-
nomialToUnivariatePolynomial univariate LIMITPS PowerSeries-
LimitPackage complexLimit limit PRIMARR2 PrimitiveArrayFunc-
tions2 map reduce scan PRIMELT PrimitiveElement primitiveEle-
ment ODEPRIM PrimitiveRatDE denomLODE ODEPRRIC Prim-
itiveRatRicDE changevar constantCoefficientRicDE denomRicDE
leadingCoefficientRicDE polyRicDE singRicDE PRINT PrintPack-
age print INTPAF PureAlgebraicIntegration palgLODE palgRDE
palgextint palgint palglimint ODEPAL PureAlgebraicLODE algD-
solve QALGSET2 QuasiAlgebraicSet2 radicalSimplify QUATCT2
QuaternionCategoryFunctions2 map QFCAT2 QuotientFieldCatego-
ryFunctions2 map REP RadicalEigenPackage eigenMatrix gram-
schmidt normalise orthonormalBasis radicalEigenvalues radicalEigen-
vector radicalEigenvectors SOLVERAD RadicalSolvePackage con-
tractSolve radicalRoots radicalSolve RADUTIL RadixUtilities radix

RANDSRC RandomNumberSource randnum reseed size RATFACT
RationalFactorize   factor DEFINTRF RationalFunctionDefiniteIn-
tegration   integrate RFFACTOR RationalFunctionFactorizer   fac-
torFraction RFFACT RationalFunctionFactor   factor INTRF Ratio-
nalFunctionIntegration extendedIntegrate infieldIntegrate internal-
Integrate limitedIntegrate LIMITRF RationalFunctionLimitPackage
complexLimit limit SIGNRF RationalFunctionSign sign SUMRF Ra-
tionalFunctionSum   sum RF RationalFunction   coerce eval main-
Variable multivariate univariate variables INTRAT RationalIntegra-
tion   extendedint infieldint integrate limitedint ODERAT Ratio-
nalLODE ratDsolve RATRET RationalRetractions   rational ratio-
nal? rationalIfCan ODERTRIC RationalRicDE changevar constant-
CoefficientRicDE polyRicDE ricDsolve singRicDE RTODETLS Ra-
tODETools genericPolynomial REALSOLV RealSolvePackage   real-
Solve solve REAL0Q RealZeroPackageQ realZeros refine REAL0 Re-
alZeroPackage midpoint midpoints realZeros refine RMCAT2 Rect-
angularMatrixCategoryFunctions2   map reduce RDIV ReducedDi-
visor   order ODERED ReduceLODE   reduceLODE REDORDER
ReductionOfOrder ReduceOrder REPDB RepeatedDoubling dou-
ble REPSQ RepeatedSquaring expt REP1 RepresentationPackage1
antisymmetricTensors createGenericMatrix permutationRepresenta-
tion symmetricTensors tensorProduct REP2 RepresentationPackage2
areEquivalent? completeEchelonBasis createRandomElement cyclic-
Submodule isAbsolutelyIrreducible?   meatAxe scanOneDimSub-
spaces split standardBasisOfCyclicSubmodule RESLATC ResolveLat-
ticeCompletion coerce RETSOL RetractSolvePackage solveRetract
SAERFFC SAERationalFunctionAlgFactor factor SEGBIND2 Seg-
mentBindingFunctions2 map SEG2 SegmentFunctions2 map SAE-
FACT SimpleAlgebraicExtensionAlgFactor   factor DFLOATSFUN
DoubleFloatSpecialFunctions   Beta Gamma airyAi airyBi besselI
besselJ besselK besselY digamma hypergeometric0F1 logGamma
polygamma SCACHE SortedCache cache clearCache enterInCache
SUP2 SparseUnivariatePolynomialFunctions2 map SPECOUT Spe-
cialOutputPackage outputAsFortran outputAsScript outputAsTex
MATSTOR StorageEfficientMatrixOperations ** copy! leftScalar-
Times! minus! plus! power! rightScalarTimes! times! STREAM1
StreamFunctions1   concat STREAM2 StreamFunctions2   map re-
duce scan STREAM3 StreamFunctions3 map STTAYLOR Stream-
TaylorSeriesOperations * + - / addiag coerce compose deriv eval
evenlambert gderiv generalLambert int integers integrate invmul-
tisect lagrange lambert lazyGintegrate lazyIntegrate mapdiv map-
mult monom multisect nlde oddintegers oddlambert power powern
recip revert STTF StreamTranscendentalFunctions ** acos acosh acot
acoth acsc acsch asec asech asin asinh atan atanh cos cosh cot coth
csc csch exp log sec sech sin sincos sinh sinhcosh tan tanh SUBRESP
SubResultantPackage primitivePart subresultantVector SYMFUNC

SymmetricFunctions symFunc SGCF SymmetricGroupCombina-
toricFunctions coleman inverseColeman listYoungTableaus makeY-
oungTableau nextColeman nextLatticePermutation nextPartition
numberOfImproperPartitions subSet unrankImproperPartitions0 un-
rankImproperPartitions1 ODESYS SystemODESolver solveInField
triangulate SYSSOLP SystemSolvePackage solve triangularSys-
tems TABLBUMP TableauxBumpers bat bat1 bumprow bumptab
bumptab1 inverse lex maxrow mr slex tab tab1 untab TANEXP
TangentExpansions tanAn tanNa tanSum TOOLSIGN ToolsFor-
Sign direction nonQsign sign DRAWCURV TopLevelDrawFunctions-
ForAlgebraicCurves draw DRAWCFUN TopLevelDrawFunctions-
ForCompiledFunctions draw makeObject recolor DRAW TopLevel-
DrawFunctions draw makeObject TOPSP TopLevelThreeSpace cre-
ateThreeSpace INTHERTR TranscendentalHermiteIntegration Her-
miteIntegrate INTTR TranscendentalIntegration expextendedint
expintegrate expintfldpoly explimitedint primextendedint primex-
tintfrac primintegrate primintegratefrac primintfldpoly primlimint-
frac primlimitedint TRMANIP TranscendentalManipulations cos2sec
cosh2sech cot2tan cot2trig coth2tanh coth2trigh csc2sin csch2sinh ex-
pand expandLog expandPower htrigs removeCosSq removeCoshSq
removeSinSq removeSinhSq sec2cos sech2cosh simplify simplifyExp
sin2csc sinh2csch tan2cot tan2trig tanh2coth tanh2trigh RDETR
TranscendentalRischDE DSPDE SPDE baseRDE expRDE prim-
RDE SOLVESER TransSolvePackageService decomposeFunc unvec-
torise SOLVETRA TransSolvePackage solve TRIMAT TriangularMa-
trixOperations LowTriBddDenomInv UpTriBddDenomInv TRIGM-
NIP TrigonometricManipulations complexElementary complexNor-
malize imag real real? trigs TUBETOOL TubePlotTools * + −
cosSinInfo cross dot loopPoints point unitVector CLIP TwoDimen-
sionalPlotClipping clip clipParametric clipWithRanges TWOFACT
TwoFactorize generalSqFr generalTwoFactor twoFactor UNIFACT
UnivariateFactorize factor factorSquareFree genFact henselFact hen-
selfact quadratic sqroot trueFactors ULS2 UnivariateLaurentSeries-
Functions2 map UPOLYC2 UnivariatePolynomialCategoryFunc-
tions2 map UPCDEN UnivariatePolynomialCommonDenominator
clearDenominator commonDenominator splitDenominator UP2 Uni-
variatePolynomialFunctions2 map UPSQFREE UnivariatePolynomi-
alSquareFree BumInSepFFE squareFree squareFreePart UPXS2 Uni-
variatePuiseuxSeriesFunctions2 map UTS2 UnivariateTaylorSeries-
Functions2 map UTSODE UnivariateTaylorSeriesODESolver mp-
sode ode ode1 ode2 stFunc1 stFunc2 stFuncN UNISEG2 Uni-
versalSegmentFunctions2 map UDPO UserDefinedPartialOrdering
getOrder largest less? more? setOrder userOrdered? UDVO UserDe-
finedVariableOrdering getVariableOrder resetVariableOrder setVari-
ableOrder VECTOR2 VectorFunctions2 map reduce scan VIEWDEF
ViewDefaultsPackage axesColorDefault lineColorDefault pointCol-

orDefault pointSizeDefault tubePointsDefault tubeRadiusDefault unitsColorDefault var1StepsDefault var2StepsDefault viewDefaults viewPosDefault viewSizeDefault viewWriteAvailable viewWriteDefault VIEW ViewportPackage coerce drawCurves graphCurves WEIER WeierstrassPreparation cfirst clikeUniv crest qqq sts2stst weierstrass WFFINTBS WildFunctionFieldIntegralBasis integralBasis listSquaredFactors

Appendix E

Operations

This appendix contains a partial list of Axiom operations with brief descriptions. For more details, use the Browse facility of HyperDoc: enter the name of the operation for which you want more information in the input area on the main Browse menu and then click on **Operations.**

Operations

Appendix F

Programs for AXIOM Images

This appendix contains the Axiom programs used to generate the images in the gallery color insert of this book. All these input files are included with the Axiom system. To produce the images on page 6 of the gallery insert, for example, issue the command:

```
)read images6
```

These images were produced on an IBM RS/6000 model 530 with a standard color graphics adapter. The smooth shaded images were made from X Window System screen dumps. The remaining images were produced with Axiom-generated PostScript output. The images were reproduced from slides made on an Agfa ChromaScript PostScript interpreter with a Matrix Instruments QCR camera.

F.1   images1.input

```
)read tknot                             Read torus knot program

torusKnot(15,17, 0.1, 6, 700)           A (15,17) torus knot
```

F.2   images2.input

These images illustrate how Newton's method converges when computing the complex cube roots of 2. Each point in the (x, y)-plane represents the complex number $x + iy$, which is given as a starting point for Newton's method. The poles in these images represent bad starting values. The flat areas are the regions of convergence to the three roots.

```
)read newton                           Read the programs from
)read vectors                          Chapter 10
f := newtonStep(x**3 - 2)              Create a Newton's iteration
                                       function for $x^3 = 2$
```

The function f^n computes n steps of Newton's method.

```
clipValue := 4                          Clip values with magnitude > 4
drawComplexVectorField(f**3, -3..3, -3..3)  The vector field for $f^3$
drawComplex(f**3, -3..3, -3..3)         The surface for $f^3$
drawComplex(f**4, -3..3, -3..3)         The surface for $f^4$
```

F.3   images3.input

```
)r tknot
for i in 0..4 repeat torusKnot(2, 2 + i/4, 0.5, 25, 250)
```

F.4   images5.input

The parameterization of the Etruscan Venus is due to George Frances.

```
venus(a,r,steps) ==
  surf := (u:DFLOAT, v:DFLOAT): Point DFLOAT +->
    cv := cos(v)
    sv := sin(v)
    cu := cos(u)
    su := sin(u)
    x := r * cos(2*u) * cv + sv * cu
    y := r * sin(2*u) * cv - sv * su
    z := a * cv
    point [x,y,z]
  draw(surf, 0..\%pi, -\%pi..\%pi, var1Steps==steps,
      var2Steps==steps, title == "Etruscan Venus")

venus(5/2, 13/10, 50)                           The Etruscan Venus
```

The Figure-8 Klein Bottle parameterization is from "Differential Geometry and Computer Graphics" by Thomas Banchoff, in *Perspectives in Mathematics,* Anniversary of Oberwolfasch 1984, Birkhäuser-Verlag, Basel, pp. 43-60.

```
klein(x,y) ==
  cx := cos(x)
  cy := cos(y)
  sx := sin(x)
  sy := sin(y)
  sx2 := sin(x/2)
  cx2 := cos(x/2)
  sq2 := sqrt(2.0@DFLOAT)
  point [cx * (cx2 * (sq2 + cy) + (sx2 * sy * cy)), \_
         sx * (cx2 * (sq2 + cy) + (sx2 * sy * cy)), \_
         -sx2 * (sq2 + cy) + cx2 * sy * cy]

draw(klein, 0..4*\%pi, 0..2*\%pi, var1Steps==50,      Figure-8 Klein bottle
     var2Steps==50,title=="Figure Eight Klein Bottle")
```

The next two images are examples of generalized tubes.

```
)read ntube
rotateBy(p, theta) ==                          Rotate a point $p$ by
  c := cos(theta)                              $\theta$ around the origin
  s := sin(theta)
  point [p.1*c - p.2*s, p.1*s + p.2*c]

bcircle t ==                                   A circle in three-space
  point [3*cos t, 3*sin t, 0]

twist(u, t) ==                                 An ellipse that twists
  theta := 4*t                                 around four times as
  p := point [sin u, cos(u)/2]                 $t$ revolves once
  rotateBy(p, theta)

ntubeDrawOpt(bcircle, twist, 0..2*\%pi, 0..2*\%pi,    Twisted Torus
            var1Steps == 70, var2Steps == 250)

twist2(u, t) ==                                Create a twisting circle
  theta := t
  p := point [sin u, cos(u)]
  rotateBy(p, theta)

cf(u,v) == sin(21*u)                           Color function with $21$ stripes

ntubeDrawOpt(bcircle, twist2, 0..2*\%pi, 0..2*\%pi,           Striped Torus
```

```
  colorFunction == cf, var1Steps == 168,
  var2Steps == 126)
```

F.5   images6.input

```
gam(x,y) ==                                      The height and color are the
  g := Gamma complex(x,y)                        real and argument parts
  point [x,y,max(min(real g, 4), -4), argument g]  of the Gamma function,
                                                 respectively.


draw(gam, -\%pi..\%pi, -\%pi..\%pi,              The Gamma Function
     title == "Gamma(x + \%i*y)", \_
     var1Steps == 100, var2Steps == 100)


b(x,y) == Beta(x,y)


draw(b, -3.1..3, -3.1 .. 3, title == "Beta(x,y)")  The Beta Function


atf(x,y) ==
  a := atan complex(x,y)
  point [x,y,real a, argument a]


draw(atf, -3.0..\%pi, -3.0..\%pi)                The Arctangent function
```

F.6   images7.input

First we look at the conformal map $z \mapsto z + 1/z$.

```
)read conformal                                  Read program for drawing
                                                 conformal maps


f z == z                                         The coordinate grid for the
                                                 complex plane
conformalDraw(f, -2..2, -2..2, 9, 9, "cartesian")  Mapping 1: Source


f z == z + 1/z                                   The map $z \mapsto z + 1/z$


conformalDraw(f, -2..2, -2..2, 9, 9, "cartesian")  Mapping 1: Target
```

The map $z \mapsto -(z+1)/(z-1)$ maps the unit disk to the right half-plane, as
shown on the Riemann sphere.

```
f z == z                                               The unit disk
```

```
riemannConformalDraw(f,0.1..0.99,0..2*\%pi,7,11,"polar")      Mapping 2: Source

f z == -(z+1)/(z-1)                             The map $x \mapsto -(z+1)/(z-1)$

riemannConformalDraw(f,0.1..0.99,0..2*\%pi,7,11,"polar")      Mapping 2: Target

riemannSphereDraw(-4..4, -4..4, 7, 7, "cartesian")      Riemann Sphere Mapping
```

F.7   images8.input

```
)read dhtri
)read tetra
drawPyramid 4                                   Sierpinsky's Tetrahedron

Sierpinsky's Tetrahedron
)read antoine
drawRings 2                                           Antoine's Necklace

Aintoine's Necklace
)read scherk
drawScherk(3,3)                                  Scherk's Minimal Surface

)read ribbonsNew
drawRibbons([x**i for i in 1..5], x=-1..1, y=0..2)         Ribbon Plot
```

F.8   conformal.input

The functions in this section draw conformal maps both on the plane and on
the Riemann sphere.

```
C := Complex DoubleFloat                              Complex Numbers
S := Segment DoubleFloat                              Draw ranges
R3 := Point DFLOAT                                    Points in 3-space
```

conformalDraw*(f, rRange, tRange, rSteps, tSteps, coord)* draws the image
of the coordinate grid under *f* in the complex plane. The grid may be given
in either polar or Cartesian coordinates. Argument *f* is the function to draw;
rRange is the range of the radius (in polar) or real (in Cartesian); *tRange* is the
range of θ (in polar) or imaginary (in Cartesian); *tSteps, rSteps*, are the number
of intervals in the r and θ directions; and *coord* is the coordinate system to use
(either `"polar"` or `"cartesian"`).

```
conformalDraw: (C -> C, S, S, PI, PI, String) -> VIEW3D
conformalDraw(f,rRange,tRange,rSteps,tSteps,coord) ==
  transformC :=                                 Function for changing an $(x,y)$
    coord = "polar" => polar2Complex              pair into a complex number
    cartesian2Complex
  cm := makeConformalMap(f, transformC)
  sp := createThreeSpace()                        Create a fresh space
  adaptGrid(sp, cm, rRange, tRange, rSteps, tSteps)   Plot the coordinate lines
  makeViewport3D(sp, "Conformal Map")             Draw the image
```

riemannConformalDraw *(f, rRange, tRange, rSteps, tSteps, coord)* draws the
image of the coordinate grid under *f* on the Riemann sphere. The grid may be
given in either polar or Cartesian coordinates. Its arguments are the same as
those for **conformalDraw**.

```
riemannConformalDraw:(C->C,S,S,PI,PI,String)->VIEW3D
riemannConformalDraw(f, rRange, tRange,
                    rSteps, tSteps, coord) ==
  transformC :=                                 Function for changing an $(x,y)$
    coord = "polar" => polar2Complex              pair into a complex number
    cartesian2Complex
  sp := createThreeSpace()                        Create a fresh space
  cm := makeRiemannConformalMap(f, transformC)
  adaptGrid(sp, cm, rRange, tRange, rSteps, tSteps)  Plot the coordinate lines
  curve(sp,[point [0,0,2.0@DFLOAT,0],point [0,0,2.0@DFLOAT,0] ])
                                                  Add an invisible point at
  makeViewport3D(sp,"Map on the Riemann Sphere")    the north pole for scaling

adaptGrid(sp, f, uRange, vRange,  uSteps, vSteps) == Plot the coordinate grid
  delU := (hi(uRange) - lo(uRange))/uSteps          using adaptive plotting for
  delV := (hi(vRange) - lo(vRange))/vSteps           coordinate lines, and draw
  uSteps := uSteps + 1; vSteps := vSteps + 1         tubes around the lines
  u := lo uRange
  for i in 1..uSteps repeat                         Draw coordinate lines in the $v$
    c := curryLeft(f,u)                              direction; curve $c$ fixes the
    cf := (t:DFLOAT):DFLOAT +-> 0                     current value of $u$
    makeObject(c,vRange::SEG Float,colorFunction==cf,
                                                      Draw the $v$ coordinate line
      space == sp, tubeRadius == .02, tubePoints == 6)
    u := u + delU
  v := lo vRange
  for i in 1..vSteps repeat                         Draw coodinate lines in the $u$
    c := curryRight(f,v)                             direction; curve $c$ fixes the
    cf := (t:DFLOAT):DFLOAT +-> 1                     current value of $v$
    makeObject(c,uRange::SEG Float,colorFunction==cf,
                                                      Draw the $u$ coordinate line
```

```
      space == sp, tubeRadius == .02, tubePoints == 6)
    v := v + delV
  void()

riemannTransform(z) ==                          Map a point in the complex
  r := sqrt norm z                              plane to the Riemann sphere
  cosTheta := (real z)/r
  sinTheta := (imag z)/r
  cp := 4*r/(4+r**2)
  sp := sqrt(1-cp*cp)
  if r>2 then sp := -sp
  point [cosTheta*cp, sinTheta*cp, -sp + 1]

cartesian2Complex(r:DFLOAT, i:DFLOAT):C ==    Convert Cartesian coordinates to
  complex(r, i)                               complex Cartesian form

polar2Complex(r:DFLOAT, th:DFLOAT):C ==       Convert polar coordinates to
  complex(r*cos(th), r*sin(th))               complex Cartesian form

makeConformalMap(f, transformC) ==            Convert complex function $f$
  (u:DFLOAT,v:DFLOAT):R3 +->                   to a mapping:
                                               (DFLOAT,DFLOAT) $mapsto$ R3
    z := f transformC(u, v)                    in the complex plane
    point [real z, imag z, 0.0@DFLOAT]

makeRiemannConformalMap(f, transformC) ==     Convert a complex function $f$
  (u:DFLOAT, v:DFLOAT):R3 +->                   to a mapping:
                                               (DFLOAT,DFLOAT) $\mapsto$ R3
    riemannTransform f transformC(u, v)        on the Riemann sphere

riemannSphereDraw: (S, S, PI, PI, String) -> VIEW3D
                                               Draw a picture of the mapping
                                               of the complex plane to
                                               the Riemann sphere
riemannSphereDraw(rRange,tRange,rSteps,tSteps,coord) ==
  transformC :=
    coord = "polar" => polar2Complex
    cartesian2Complex
  grid := (u:DFLOAT, v:DFLOAT): R3 +->                  Coordinate grid function
    z1 := transformC(u, v)
    point [real z1, imag z1, 0]
  sp := createThreeSpace()                              Create a fresh space
  adaptGrid(sp, grid, rRange, tRange, rSteps, tSteps)     Draw the flat grid
  connectingLines(sp,grid,rRange,tRange,rSteps,tSteps)
  makeObject(riemannSphere,0..2*\%pi,0..\%pi,space==sp)     Draw the sphere
  f := (z:C):C +-> z
```

```
  cm := makeRiemannConformalMap(f, transformC)
  adaptGrid(sp, cm, rRange, tRange, rSteps, tSteps)        Draw the sphere grid
  makeViewport3D(sp, "Riemann Sphere")

connectingLines(sp,f,uRange,vRange,uSteps,vSteps) ==
                                              Draw the lines that connect
  delU := (hi(uRange) - lo(uRange))/uSteps        the points in the complex
  delV := (hi(vRange) - lo(vRange))/vSteps        plane to the north pole
  uSteps := uSteps + 1; vSteps := vSteps + 1      of the Riemann sphere
  u := lo uRange
  for i in 1..uSteps repeat                       For each u
    v := lo vRange
    for j in 1..vSteps repeat                     For each v
      p1 := f(u,v)
      p2 := riemannTransform complex(p1.1, p1.2)  Project p1 onto the sphere
      fun := lineFromTo(p1,p2)                     Create a line function
      cf := (t:DFLOAT):DFLOAT +-> 3
      makeObject(fun, 0..1,space==sp,tubePoints==4, Draw the connecting line
                tubeRadius==0.01,colorFunction==cf)
      v := v + delV
    u := u + delU
  void()

riemannSphere(u,v) ==                              A sphere sitting on the
  sv := sin(v)                                     complex plane, with radius 1
  0.99@DFLOAT*(point [cos(u)*sv,sin(u)*sv,cos(v),0.0@DFLOAT])+
    point [0.0@DFLOAT, 0.0@DFLOAT, 1.0@DFLOAT, 4.0@DFLOAT]

lineFromTo(p1, p2) ==                              Create a line function
  d := p2 - p1                                     that goes from p1 to p2
  (t:DFLOAT):Point DFLOAT +->
    p1 + t*d
```

F.9   tknot.input

Create a (p,q) torus-knot with radius r around the curve. The formula was
derived by Larry Lambe.

```
)read ntube
torusKnot: (DFLOAT, DFLOAT, DFLOAT, PI, PI) -> VIEW3D
torusKnot(p, q ,r, uSteps, tSteps) ==
  knot := (t:DFLOAT):Point DFLOAT +->             Function for the torus knot
    fac := 4/(2.2@DFLOAT-sin(q*t))
    fac * point [cos(p*t), sin(p*t), cos(q*t)]
```

```
  circle := (u:DFLOAT, t:DFLOAT): Point DFLOAT +->  The cross section
    r * point [cos u, sin u]
  ntubeDrawOpt(knot, circle, 0..2*\%pi, 0..2*\%pi,
                                              Draw the circle around the knot
              var1Steps == uSteps, var2Steps == tSteps)
```

F.10   ntube.input

The functions in this file create generalized tubes (also known as generalized
cylinders). These functions draw a 2-d curve in the normal planes around a 3-d
curve.

```
R3 := Point DFLOAT                                      Points in 3-Space
R2 := Point DFLOAT                                      Points in 2-Space
S := Segment Float                                      Draw ranges
                                        Introduce types for functions for:
ThreeCurve := DFLOAT -> R3              --the space curve function
TwoCurve := (DFLOAT, DFLOAT) -> R2      --the plane curve function
Surface := (DFLOAT, DFLOAT) -> R3       --the surface function
                                        Frenet frames define a
FrenetFrame :=                          coordinate system around a
   Record(value:R3,tangent:R3,normal:R3,binormal:R3)
                                        point on a space curve
frame: FrenetFrame                      The current Frenet frame
                                        for a point on a curve
```

ntubeDraw(*spaceCurve, planeCurve, $u_0..u_1$, $t_0..t_1$*) draws *planeCurve* in the
normal planes of *spaceCurve*. The parameter $u_0..u_1$ specifies the parameter
range for *planeCurve* and $t_0..t_1$ specifies the parameter range for *spaceCurve*.
Additionally, the plane curve function takes a second parameter: the current
parameter of *spaceCurve*. This allows the plane curve to change shape as it goes
around the space curve. See F.4 for an example of this.

```
ntubeDraw: (ThreeCurve,TwoCurve,S,S) -> VIEW3D
ntubeDraw(spaceCurve,planeCurve,uRange,tRange) ==
  ntubeDrawOpt(spaceCurve, planeCurve, uRange, \_
               tRange, []\$List DROPT)

ntubeDrawOpt: (ThreeCurve,TwoCurve,S,S,List DROPT)
    -> VIEW3D
ntubeDrawOpt(spaceCurve,planeCurve,uRange,tRange,l) ==
                                              This function is similar
                                              to ntubeDraw, but takes
```

```
    delT:DFLOAT := (hi(tRange) - lo(tRange))/10000  optional parameters that it
    oldT:DFLOAT := lo(tRange) - 1                    passes to the draw command
    fun := ngeneralTube(spaceCurve,planeCurve,delT,oldT)
    draw(fun, uRange, tRange, l)
```

nfrenetFrame*(c, t, delT)* numerically computes the Frenet frame about the
curve c at t. Parameter *delT* is a small number used to compute derivatives.

```
nfrenetFrame(c, t, delT) ==
  f0 := c(t)
  f1 := c(t+delT)
  t0 := f1 - f0                          The tangent
  n0 := f1 + f0
  b := cross(t0, n0)                     The binormal
  n := cross(b,t0)                       The normal
  ln := length n
  lb := length b
  ln = 0 or lb = 0 =>
      error "Frenet Frame not well defined"
  n := (1/ln)*n                          Make into unit length vectors
  b := (1/lb)*b
  [f0, t0, n, b]\$FrenetFrame
```

ngeneralTube*(spaceCurve, planeCurve,delT, oltT)* creates a function that can
be passed to the system axiomFundraw command. The function is a parame-
terized surface for the general tube around *spaceCurve*. *delT* is a small number
used to compute derivatives. *oldT* is used to hold the current value of the t
parameter for *spaceCurve*. This is an efficiency measure to ensure that frames
are only computed once for each value of t.

```
ngeneralTube: (ThreeCurve, TwoCurve, DFLOAT, DFLOAT) -> Surface
ngeneralTube(spaceCurve, planeCurve, delT, oldT) ==
  free frame                             Indicate that $frame$ is global
  (v:DFLOAT, t: DFLOAT): R3 +->
    if (t $\sim$= oldT) then             If not already computed
      frame := nfrenetFrame(spaceCurve, t, delT)  compute new frame
      oldT := t
    p := planeCurve(v, t)
    frame.value + p.1*frame.normal + p.2*frame.binormal
                                         Project $p$ into the normal plane
```

F.11   dhtri.input

Create affine transformations (DH matrices) that transform a given triangle into
another.

```
tri2tri: (List Point DFLOAT, List Point DFLOAT) -> DHMATRIX(DFLOAT)
                                            Compute a DHMATRIX that
tri2tri(t1, t2) ==                          transforms $t1$ to $t2,$ where
  n1 := triangleNormal(t1)                  $t1$ and $t2$ are the vertices
  n2 := triangleNormal(t2)                  of two triangles in 3-space
  tet2tet(concat(t1, n1), concat(t2, n2))


tet2tet: (List Point DFLOAT, List Point DFLOAT) -> DHMATRIX(DFLOAT)
                                            Compute a DHMATRIX that
tet2tet(t1, t2) ==                          transforms $t1$ to $t2,$
  m1 := makeColumnMatrix t1                 where $t1$ and $t2$ are the
  m2 := makeColumnMatrix t2                 vertices of two tetrahedrons
  m2 * inverse(m1)                          in 3-space

makeColumnMatrix(t) ==                      Put the vertices of a tetra-
  m := new(4,4,0)\$DHMATRIX(DFLOAT)         hedron into matrix form
  for x in t for i in 1..repeat
    for j in 1..3 repeat
      m(j,i) := x.j
    m(4,i) := 1
  m

triangleNormal(t) ==                        Compute a vector normal to
  a := triangleArea t                       the given triangle, whose
  p1 := t.2 - t.1                           length is the square root
  p2 := t.3 - t.2                           of the area of the triangle
  c := cross(p1, p2)
  len := length(c)
  len = 0 => error "degenerate triangle!"
  c := (1/len)*c
  t.1 + sqrt(a) * c

triangleArea t ==                           Compute the area of a
  a := length(t.2 - t.1)                    triangle using Heron's
  b := length(t.3 - t.2)                    formula
  c := length(t.1 - t.3)
  s := (a+b+c)/2
  sqrt(s*(s-a)*(s-b)*(s-c))
```

F.12   tetra.input

```
)set expose add con DenavitHartenbergMatrix      Bring DH matrices into the
                                                 environment
x1:DFLOAT := sqrt(2.0@DFLOAT/3.0@DFLOAT)         Set up the coordinates of the
```

```
x2:DFLOAT := sqrt(3.0@DFLOAT)/6                      corners of the tetrahedron.

p1 := point [-0.5@DFLOAT, -x2, 0.0@DFLOAT]           Some needed points
p2 := point [0.5@DFLOAT, -x2, 0.0@DFLOAT]
p3 := point [0.0@DFLOAT, 2*x2, 0.0@DFLOAT]
p4 := point [0.0@DFLOAT, 0.0@DFLOAT, x1]

baseTriangle  := [p2, p1, p3]                        The base of the tetrahedron

mt  := [0.5@DFLOAT*(p2+p1), 0.5@DFLOAT*(p1+p3), 0.5@DFLOAT*(p3+p2)]
                                                     The middle triangle inscribed
                                                     in the base of the tetrahedron
bt1 := [mt.1, p1, mt.2]                              The bases of the triangles of
bt2 := [p2, mt.1, mt.3]                              the subdivided tetrahedron
bt3 := [mt.2, p3, mt.3]
bt4 := [0.5@DFLOAT*(p2+p4), 0.5@DFLOAT*(p1+p4), 0.5@DFLOAT*(p3+p4)]

tt1 := tri2tri(baseTriangle, bt1)                    Create the transformations
tt2 := tri2tri(baseTriangle, bt2)                    that bring the base of the
tt3 := tri2tri(baseTriangle, bt3)                    tetrahedron to the bases of
tt4 := tri2tri(baseTriangle, bt4)                    the subdivided tetrahedron

drawPyramid(n) ==                                    Draw a Sierpinsky tetrahedron
  s := createThreeSpace()                            with $n$ levels of recursive
  dh := rotatex(0.0@DFLOAT)                           subdivision
  drawPyramidInner(s, n, dh)
  makeViewport3D(s, "Sierpinsky Tetrahedron")

drawPyramidInner(s, n, dh) ==                        Recursively draw a Sierpinsky
  n = 0 => makeTetrahedron(s, dh, n)                 tetrahedron
  drawPyramidInner(s, n-1, dh * tt1)                 Draw the 4 recursive pyramids
  drawPyramidInner(s, n-1, dh * tt2)
  drawPyramidInner(s, n-1, dh * tt3)
  drawPyramidInner(s, n-1, dh * tt4)

makeTetrahedron(sp, dh, color) ==                    Draw a tetrahedron into the
  w1 := dh*p1                                         given space with the given
  w2 := dh*p2                                         color, transforming it by
  w3 := dh*p3                                         the given DH matrix
  w4 := dh*p4
  polygon(sp, [w1, w2, w4])
  polygon(sp, [w1, w3, w4])
  polygon(sp, [w2, w3, w4])
  void()
```

F.13 antoine.input

Draw Antoine's Necklace. Thank you to Matthew Grayson at IBM's T.J Watson
Research Center for the idea.

```
)set expose add con DenavitHartenbergMatrix          Bring DH matrices into
                                                     the environment
torusRot: DHMATRIX(DFLOAT)                           The  transformation for
                                                     drawing a sub ring


drawRings(n) ==                                      Draw Antoine's Necklace with $n$
  s := createThreeSpace()                            levels of recursive subdivision
  dh:DHMATRIX(DFLOAT) := identity()                  The number of subrings is $10^n$
  drawRingsInner(s, n, dh)                           Do the real work
  makeViewport3D(s, "Antoine's Necklace")
```

In order to draw Antoine rings, we take one ring, scale it down to a smaller
size, rotate it around its central axis, translate it to the edge of the larger ring
and rotate it around the edge to a point corresponding to its count (there are
10 positions around the edge of the larger ring). For each of these new rings
we recursively perform the operations, each ring becoming 10 smaller rings.
Notice how the **DHMATRIX** operations are used to build up the proper matrix
composing all these transformations.

```
drawRingsInner(s, n, dh) ==                          Recursively draw Antoine's
  n = 0 =>                                           Necklace
    drawRing(s, dh)
    void()
  t := 0.0@DFLOAT                                    Angle around ring
  p := 0.0@DFLOAT                                    Angle of subring from plane
  tr := 1.0@DFLOAT                                   Amount to translate subring
  inc := 0.1@DFLOAT                                  The translation increment
  for i in 1..10 repeat                              Subdivide into 10 linked rings
    tr := tr + inc
    inc := -inc
    dh' := dh*rotatez(t)*translate(tr,0.0@DFLOAT,0.0@DFLOAT)*
                                                     Transform ring in center
                                                     to a link
          rotatey(p)*scale(0.35@DFLOAT, 0.48@DFLOAT, 0.4@DFLOAT)
    drawRingsInner(s, n-1, dh')
    t := t + 36.0@DFLOAT
    p := p + 90.0@DFLOAT
  void()
```

```
drawRing(s, dh) ==                      Draw a single ring into
  free torusRot                         the given subspace,
  torusRot := dh                        transformed by the given
                                        DHMATRIX
  makeObject(torus, 0..2*\%pi, 0..2*\%pi, var1Steps == 6,
            space == s, var2Steps == 15)

torus(u ,v) ==                          Parameterization of a torus,
  cu := cos(u)/6                        transformed by the
                                        DHMATRIX in $torusRot.$
  torusRot*point [(1+cu)*cos(v),(1+cu)*sin(v),(sin u)/6]
```

F.14   scherk.input

Scherk's minimal surface, defined by: $e^z \cos(x) = \cos(y)$. See: *A Comprehensive Introduction to Differential Geometry,* Vol. 3, by Michael Spivak, Publish Or Perish, Berkeley, 1979, pp. 249-252.

```
(xOffset, yOffset):DFLOAT                 Offsets for a single piece
                                          of Scherk's minimal surface

drawScherk(m,n) ==                        Draw Scherk's minimal surface
  free xOffset, yOffset                   on an $m$ by $n$ patch
  space := createThreeSpace()
  for i in 0..m-1 repeat
    xOffset := i*\%pi
    for j in 0 .. n-1 repeat
      rem(i+j, 2) = 0 => 'iter             Draw only odd patches
      yOffset := j*\%pi
      drawOneScherk(space)                Draw a patch
  makeViewport3D(space, "Scherk's Minimal Surface")

scherk1(u,v) ==                           The first patch that makes
  x := cos(u)/exp(v)                      up a single piece of
  point [xOffset + acos(x), yOffset + u, v, abs(v)]  Scherk's minimal surface

scherk2(u,v) ==                           The second patch
  x := cos(u)/exp(v)
  point [xOffset - acos(x), yOffset + u, v, abs(v)]

scherk3(u,v) ==                           The third patch
  x := exp(v) * cos(u)
  point [xOffset + u, yOffset + acos(x), v, abs(v)]
```

```
scherk4(u,v) ==                                     The fourth patch
  x := exp(v) * cos(u)
  point [xOffset + u, yOffset - acos(x), v, abs(v)]

drawOneScherk(s) ==                                 Draw the surface by
                                                    breaking it into four
                                                    patches and then drawing
                                                    the patches
  makeObject(scherk1,-\%pi/2..\%pi/2,0..\%pi/2,space==s,
             var1Steps == 28, var2Steps == 28)
  makeObject(scherk2,-\%pi/2..\%pi/2,0..\%pi/2,space==s,
             var1Steps == 28, var2Steps == 28)
  makeObject(scherk3,-\%pi/2..\%pi/2,-\%pi/2..0,space==s,
             var1Steps == 28, var2Steps == 28)
  makeObject(scherk4,-\%pi/2..\%pi/2,-\%pi/2..0,space==s,
             var1Steps == 28, var2Steps == 28)
  void()
```

Appendix G

Glossary

Glossary

!
(syntax) Suffix character for destructive operations.

,
(syntax) a separator for items in a *tuple*, for example, to separate arguments of a function $f(x, y)$.

=>
(syntax) the expression $a => b$ is equivalent to $if\ a\ then\ exit\ b$.

?
1. *(syntax)* a suffix character for Boolean-valued **function** names, for example, **odd?**. 2. Prefix character for "optional" pattern variables. For example, the pattern $f(x + y)$ does not match the expression $f(7)$, but $f(?x + y)$ does, with x matching 0 and y matching 7. 3. The special type **?** means *don't care*. For example, the declaration: $x : Polynomial?$ means that values assigned to x must be polynomials over an arbitrary *underlying domain*.

abstract datatype
a programming language principle used in Axiom where a datatype definition has defined in two parts: (1) a *public* part describing a set of exports, principally operations that apply to objects of that type, and (2) a *private* part describing the implementation of the datatype usually in terms of a *representation* for objects of the type. Programs that create and otherwise manipulate objects of the type may only do so through its exports. The representation and other implementation information is specifically hidden.

abstraction
described functionally or conceptually without regard to implementation.

accuracy
the degree of exactness of an approximation or measurement. In computer al-

gebra systems, computations are typically carried out with complete accuracy using integers or rational numbers of indefinite size. Domain **Float** provides a function **precision** to change the precision for floating-point computations. Computations using **DoubleFloat** have a fixed precision but uncertain accuracy.

add-chain
a hierarchy formed by domain extensions. If domain A extends domain B and domain B extends domain C, then A has *add-chain B-C*.

aggregate
a data structure designed to hold multiple values. Examples of aggregates are **List**, **Set**, **Matrix** and **Bits**.

AKCL
Austin Kyoto Common LISP, a version of *KCL* produced by William Schelter, Austin, Texas.

algorithm
a step-by-step procedure for a solution of a problem; a program

ancestor
(of a domain or category) a category that is a *parent*, or a *parent* of a *parent*, and so on. See a **Cross Reference** page of a constructor in Browse.

application
(syntax) an expression denoting "application" of a function to a set of *argument* parameters. Applications are written as a *parameterized form*. For example, the form $f(x, y)$ indicates the "application of the function f to the tuple of arguments x and y." See also *evaluation* and *invocation*.

apply
See *application*.

argument
1. (actual argument) a value passed to a function at the time of a function call; also called an *actual parameter*. 2. (formal argument) a variable used in the definition of a function to denote the actual argument passed when the function is called.

arity
1. (function) the number of arguments. 2. (operator or operation) corresponds to the arity of a function implementing the operator or operation.

assignment
(syntax) an expression of the form $x := e$, meaning "assign the value of e to x." After *evaluation*, the *variable* x points to an object obtained by evaluating the expression e. If x has a *type* as a result of a previous *declaration*, the object assigned to x must have that type. The interpreter must often coerce the value of e to make that happen. For example, the expression $x : Float := 11$ first declares x to be a float, then forces the interpreter to coerce the integer 11 to 11.0 in order to assign a floating-point value to x.

attribute

a name or functional form denoting *any* useful computational or mathematical property. For example, **commutative(" ∗ ")** asserts that ∗ is commutative. Also, **finiteAggregate** is used to assert that an aggregate has a finite number of immediate components.

basis

(algebra) S is a basis of a module *M* over a *ring* if *S* generates *M*, and *S* is linearly independent.

benefactor

(of a given domain) a domain or package that the given domain explicitly references (for example, calls functions from) in its implementation. See a **Cross Reference** page of a constructor in Browse.

binary

operation or function with *arity* 2.

binding

the association of a variable with properties such as *value* and *type.* The top-level *environment* in the interpreter consists of bindings for all user variables and functions. When a *function* is applied to arguments, a local environment of bindings is created, one for each formal *argument* and *local variable.*

block

(syntax) a control structure where expressions are sequentially evaluated.

body

a *function body* or *loop body.*

boolean

objects denoted by the literals `true` and `false`; elements of domain **Boolean**. See also **Bits**.

built-in function

a *function* in the standard Axiom library. Contrast *user function.*

v

cache

1. (noun) a mechanism for immediate retrieval of previously computed data. For example, a function that does a lengthy computation might store its values in a *hash table* using the function argument as the key. The hash table then serves as a cache for the function (see also `)set function cache`). Also, when recurrence relations that depend upon n previous values are compiled, the previous n values are normally cached (use `)set functions recurrence` to change this). 2. (verb) to save values in a cache.

capsule

the part of the body of a *domain constructor* that defines the functions implemented by the constructor.

case

(syntax) an operator used to evaluate code conditionally based on the branch of a **Union**. For example, if value u is $Union(Integer,"failed")$, the conditional expression $if u case Integer then A else B$ evaluates A if u is an integer and B otherwise.

Category

the distinguished object denoting the type of a category; the class of all categories.

category

(basic concept) types denoting classes of domains. Examples of categories are **Ring** ("the class of all rings") and **Aggregate** ("the class of all aggregates"). Categories form a hierarchy (formally, a directed acyclic graph) with the distinquished category **Type** at the top. Each category inherits the properties of all its ancestors. Categories optionally provide "default definitions" for operations they export. Categories are defined in Axiom by functions called category constructors. Technically, a category designates a class of domains with common operations and attributes but usually with different functions and representations for its constituent objects. Categories are always defined using the Axiom library language (see also *category extension*). See also file **catdef.spad** for definitions of basic algebraic categories in Axiom, **aggcat.spad** for data structure

category constructor

a function that creates categories, described by an abstract datatype in the Axiom programming language. For example, the category constructor **Module** is a function that takes a domain parameter R and creates the category "modules over R."

category extension

A category A *directly extends* a category B if its definition has the form $A ==$ $B with...$ or $A == Join(..., B, ...)$. In this case, we also say that B is the *parent* of A. We say that a category A extends B if B is an *ancestor* of A. A category A may also directly extend B if B appears in a conditional expression within the `Exports` part of the definition to the right of a `with`. See, for example, file **catdef.spad** for definitions of the algebra categories in Axiom, **aggcat.spad** for data structure categories.

category hierarchy

hierarchy formed by category extensions. The root category is **Type**. A category can be defined as a *Join* of two or more categories so as to have multiple parents. Categories may also be parameterized so as to allow conditional inheritance.

character

1. an element of a character set, as represented by a keyboard key. 2. a component of a string. For example, the 1st element of the string $"hello there"$ is the character h.

client

(of a given domain) any domain or package that explicitly calls functions from

the given domain. See a **Cross Reference** page of a constructor in Browse.

coercion

an automatic transformation of an object of one *type* to an object of a similar or desired target type. In the interpreter, coercions and retractions are done automatically by the interpreter when a type mismatch occurs. Compare *conversion.*

comment

textual remarks imbedded in code. Comments are preceded by a double dash (`--`). For Axiom library code, stylized comments for on-line documentation are preceded by two plus signs (`++`).

Common LISP

A version of *LISP* adopted as an informal standard by major users and suppliers of LISP.

compile-time

the time when category or domain constructors are compiled. Contrast *run-time.*

compiler

a program that generates low-level code from a higher-level source language. Axiom has three compilers. A *graphics compiler* converts graphical formulas to a compiled subroutine so that points can be rapidly produced for graphics commands. An *interpreter compiler* optionally compiles user functions when first invoked (use `)set functions compile` to turn this feature on). A *library compiler* compiles all constructors (available on an "as-is" basis for Release 1).

computational object

In Axiom, domains are objects. This term is used to distinguish the objects that are members of domains rather than the domains themselves.

conditional

a *control structure* of the form *ifAthenBelseC*. The *evaluation* of A produces `true` or `false`. If `true`, B evaluates to produce a value; otherwise C evaluates to produce a value. When the value is not required, the *elseC* part can be omitted.

constant

(syntax) a reserved word used in signatures in Axiom programming language to signify that an operation always returns the same value. For example, the signature $0 : constant- > \$$ in the source code of **AbelianMonoid** tells the Axiom compiler that 0 is a constant so that suitable optimizations might be performed.

constructor

a *function* that creates a *category*, *domain*, or *package.*

continuation

when a line of a program is so long that it must be broken into several lines, then all but the first line are called *continuation lines.* If such a line is given

interactively, then each incomplete line must end with an underscore.

control structure
program structures that can specify a departure from normal sequential execution. Axiom has four kinds of control structures: blocks, *case* statements, conditionals, and loops.

conversion
the transformation of an object of one *type* to one of another type. Conversions that can be performed automatically by the interpreter are called coercions. These happen when the interpreter encounters a type mismatch and a similar or declared target type is needed. In general, the user must use the infix operation :: to cause this transformation.

copying semantics
the programming language semantics used in PASCAL but *not* in Axiom. See also *pointer semantics* for details.

data structure
a structure for storing data in the computer. Examples are lists and hash tables.

datatype
equivalent to *domain* in Axiom.

declaration
(syntax) an expression of the form $x : T$ where T is some *type*. A declaration forces all values assigned to x to be of that type. If a value is of a different type, the interpreter will try to coerce the value to type T. Declarations are necessary in case of ambiguity or when a user wants to introduce an unexposed domain.

default definition
a function defined by a *category*. Such definitions appear in category definitions of the form
$C : Category == T add I$
in an optional implementation part I to the right of the keyword *add*.

default package
an optional *package* of functions associated with a category. Such functions are necessarily defined in terms of other operations exported by the category.

definition
(syntax) 1. An expression of the form $f(a) == b$ defining function f with formal arguments a and *body b*; equivalent to the statement $f == (a) + - > b$. 2. An expression of the form $a == b$ where a is a *symbol*, equivalent to $a() == b$. See also *macro* where a similar substitution is done at *parse* time.

delimiter
a *character* that marks the beginning or end of some syntactically correct unit in the language, for example, " for strings, blanks for identifiers.

dependent
(of a given constructor) another constructor that mentions the given constructor as an argument or among the types of an exported operation. See a **Cross**

Reference page of a constructor in Browse.

destructive operation
An operation that changes a component or structure of a value. In Axiom, destructive operations have names ending with an exclamation mark (!). For example, domain **List** has two operations to reverse the elements of a list, one named **reverse** that returns a copy of the original list with the elements reversed, another named **reverse** that reverses the elements *in place,* thus destructively changing the original list.

documentation
1. on-line or hard-copy descriptions of Axiom; 2. text in library code preceded by ++ comments as opposed to general comments preceded by --.

domain
(basic concept) a domain corresponds to the usual notion of datatypes. Examples of domains are **List Float** ("lists of floats"), **Fraction Polynomial Integer** ("fractions of polynomials of integers"), and **Matrix Stream CardinalNumber** ("matrices of infinite streams of cardinal numbers"). The term *domain* actually abbreviates *domain of computation.* Technically, a domain denotes a class of objects, a class of operations for creating and otherwise manipulating these objects, and a class of attributes describing computationally useful properties. Domains may also define functions for its exported operations, often in terms of some *representation* for the objects. A domain itself is an *object* created by a *function* called a *domain constructor.* The types of the exported operations of a domain are arbitary; this gives rise to a special class of domains called packages.

domain constructor
a function that creates domains, described by an abstract datatype in the Axiom programming language. Simple domains like **Integer** and **Boolean** are created by domain constructors with no arguments. Most domain constructors take one or more parameters, one usually denoting an *underlying domain.* For example, the domain **Matrix(R)** denotes "matrices over R." Domains **Mapping**, **Record**, and **Union** are primitive domains. All other domains are written in the Axiom programming language and can be modified by users with access to the library source code and the library compiler.

domain extension
a domain constructor A is said to *extend* a domain constructor B if A's definition has the form $A == B$add.... This intuitively means "functions not defined by A are assumed to come from B." Successive domain extensions form add-chains affecting the search order for functions not implemented directly by the domain during *dynamic lookup.*

dot notation
using an infix dot (.) for the operation **elt**. If u is the list $[7, 4, -11]$ then both $u(2)$ and $u.2$ return 4. Dot notation nests to the left: $f.g.h$ is equivalent to $(f.g).h$.

dynamic
that which is done at *run-time* as opposed to *compile-time*. For example, the interpreter may build a domain "matrices over integers" dynamically in response to user input. However, the compilation of all functions for matrices and integers is done during *compile-time*. Constrast *static*.

dynamic lookup
In Axiom, a *domain* may or may not explicitly provide *function* definitions for all its exported operations. These definitions may instead come from domains in the *add-chain* or from default packages. When a function call is made for an operation in the domain, up to five steps are carried out.

1. If the domain itself implements a function for the operation, that function is returned.

2. Each of the domains in the *add-chain* are searched; if one of these domains implements the function, that function is returned.

3. Each of the default packages for the domain are searched in order of the *lineage*. If any of the default packages implements the function, the first one found is returned.

4. Each of the default packages for each of the domains in the *add-chain* are searched in the order of their *lineage*. If any of the default packages implements the function, the first one found is returned.

5. If all of the above steps fail, an error message is reported.

empty
the unique value of objects with type **Void**.

environment
a set of bindings.

evaluation
a systematic process that transforms an *expression* into an object called the *value* of the expression. Evaluation may produce side effects.

exit
(reserved word) an *operator* that forces an exit from the current *block*. For example, the block $(a := 1; if i > 0 then exit a; a := 2)$ will prematurely exit at the second statement with value 1 if the value of i is greater than zero. See `=>` for an alternate syntax.

explicit export
1. (of a domain D) any *attribute*, *operation*, or *category* explicitly mentioned in the *type* exports part E for the domain constructor definition $D : E == I$ 2. (of a category C) any *attribute*, *operation*, or *category* explicitly mentioned in the *type* specification part E for the category constructor definition $C : Category == E$

export
explicit export or *implicit export* of a domain or category

expose
some constructors are *exposed*, others *unexposed*. Exposed domains and packages are recognized by the interpreter. Use `)set expose` to control what is exposed. Unexposed constructors will appear in Browse prefixed by a star ("*").

expression
1. any syntactically correct program fragment. 2. an element of domain **Expression**.

extend
see *category extension* or *domain extension*.

field
(algebra) a *domain* that is a *ring* where every non-zero element is invertible and where $xy = yx$; a member of category **Field**. For a complete list of fields, click on **Domains** under **Cross Reference** for **Field** in Browse.

file
1. a program or collection of data stored on disk, tape or other medium. 2. an object of a **File** domain.

float
a floating-point number with user-specified precision; an element of domain **Float**. Floats are literals written either without an exponent (for example, 3.1416), or with an exponent (for example, $3.12E - 12$). Use function *precision* to change the precision of the mantissa (20 digits by default). See also *small float*.

formal parameter
(of a function) an identifier bound to the value of an actual *argument* on *invocation*. In the function definition $f(x, y) == u$, for example, x and y are the formal parameters.

frame
the basic unit of an interactive session; each frame has its own *step number*, *environment*, and *history*. In one interactive session, users can create and drop frames, and have several active frames simultaneously.

free
(syntax) A keyword used in user-defined functions to declare that a variable is a *free variable* of that function. For example, the statement $free x$ declares the variable x within the body of a function f to be a free variable in f. Without such a declaration, any variable x that appears on the left-hand side of an assignment before it is referenced is regarded as a *local variable* of that function. If the intention of the assignment is to give a value to a *global variable* x, the body of that function must contain the statement $free x$. A variable that is a parameter to the function is always local.

free variable

(of a function) a variable that appears in a body of a function but is not bound by that function. Contrast with *local variable*.

function
implementation of *operation*. A function takes zero or more *argument* parameters and produces a single return value. Functions are objects that can be passed as parameters to functions and can be returned as values of functions. Functions can also create other functions (see also **InputForm**). See also *application* and *invocation*. The terms *operation* and *function* are distinct notions in Axiom. An operation is an abstraction of a function, described by a *name* and a *signature*. A function is created by providing an implementation of that operation by Axiom code. Consider the example of defining a user-function $fact$ to compute the **factorial** of a nonnegative integer. The Axiom statement $fact : Integer-> Integer$ describes the operation, whereas the statement $fact(n) = reduce(*, [1..n])$ defines the function. See also *generic function*.

function body
the part of a *function*'s definition that is evaluated when the function is called at *run-time*; the part of the function definition to the right of the **==**.

garbage collection
a system function that automatically recycles memory cells from the *heap*. Axiom is built upon *Common LISP* that provides this facility.

garbage collector
a mechanism for reclaiming storage in the *heap*.

Gaussian
a complex-valued expression, for example, one with both a real and imaginary part; a member of a **Complex** domain.

generic function
the use of one function to operate on objects of different types. One might regard Axiom as supporting generic operations but not generic functions. One operation $+ : (D, D)-> D$ exists for adding elements in a ring; each ring however provides its own type-specific function for implementing this operation.

global variable
A variable that can be referenced freely by functions. In Axiom, all top-level user-defined variables defined during an interactive user session are global variables. Axiom does not allow *fluid variables*, that is, variables bound by a function f that can be referenced by functions that f calls.

Gröbner basis
(algebra) a special basis for a polynomial ideal that allows a simple test for membership. It is useful in solving systems of polynomial equations.

group
(algebra) a monoid where every element has a multiplicative inverse.

hash table
a data structure designed for fast lookup of information stored under "keys".

A hash table consists of a set of *entries*, each of which associates a *key* with a *value*. Finding the object stored under a key can be fast for a large number of entries since keys are *hashed* into numerical codes for fast lookup.

heap
1. an area of storage used by data in programs. For example, Axiom will use the heap to hold the partial results of symbolic computations. When cancellations occur, these results remain in the heap until garbage collected. 2. an object of a **Heap** domain.

history
a mechanism that records input and output data for an interactive session. Using the history facility, users can save computations, review previous steps of a computation, and restore a previous interactive session at some later time. For details, issue the system command *)history ?* to the interpreter. See also *frame*.

ideal
(algebra) a subset of a ring that is closed under addition and multiplication by arbitrary ring elements; thus an ideal is a module over the ring.

identifier
(syntax) an Axiom name; a *literal* of type **Symbol**. An identifier begins with an alphabetical character, %, ?, or !, and may be followed by any of these or digits. Certain distinguished reserved words are not allowed as identifiers but have special meaning in Axiom.

immutable
an object is immutable if it cannot be changed by an *operation*; it is not a mutable object. Algebraic objects are generally immutable: changing an algebraic expression involves copying parts of the original object. One exception is an object of type **Matrix**. Examples of mutable objects are data structures such as those of type **List**. See also *pointer semantics*.

implicit export
(of a domain or category) any exported *attribute* or *operation* or *category* that is not an *explicit export*. For example, **Monoid** and ***** are implicit exports of **Ring**.

index
1. a variable that counts the number of times a *loop* is repeated. 2. the "address" of an element in a data structure (see also category **LinearAggregate**).

infix
(syntax) an *operator* placed between two operands; also called a *binary operator*. For example, in the expression $a + b$, $+$ is the infix operator. An infix operator may also be used as a *prefix*. Thus $+(a, b)$ is also permissible in the Axiom language. Infix operators have a *precedence* relative to one another.

input area
a rectangular area on a HyperDoc screen into which users can enter text.

instantiate
to build a *category*, *domain*, or *package* at run-time.

integer
a *literal* object of domain **Integer**, the class of integers with an unbounded number of digits. Integer literals consist of one or more consecutive digits (0-9) with no embedded blanks. Underscores can be used to separate digits in long integers if desirable.

interactive
a system where the user interacts with the computer step-by-step.

interpreter
the part of Axiom responsible for handling user input during an interactive session. The interpreter parses the user's input expression to create an expression tree, then does a bottom-up traversal of the tree. Each subtree encountered that is not a value consists of a root node denoting an operation name and one or more leaf nodes denoting operands. The interpreter resolves type mismatches and uses type-inferencing and a library database to determine appropriate types for the operands and the result, and an operation to be performed. The interpreter next builds a domain to perform the indicated operation, and invokes a function from the domain to compute a value. The subtree is then replaced by that value and the process continues. Once the entire tree has been processed, the value replacing the top node of the tree is displayed back to the user as the value of the expression.

invocation
(of a function) the run-time process involved in evaluating a *function application*. This process has two steps. First, a local *environment* is created where formal arguments are locally bound by *assignment* to their respective actual *argument*. Second, the *function body* is evaluated in that local environment. The evaluation of a function is terminated either by completely evaluating the function body or by the evaluation of a `return` expression.

iteration
repeated evaluation of an expression or a sequence of expressions. Iterations use the reserved words `for`, `while`, and `repeat`.

Join
a primitive Axiom function taking two or more categories as arguments and producing a category containing all of the operations and attributes from the respective categories.

KCL
Kyoto Common LISP, a version of *Common LISP* that features compilation of LISP into the *C* Programming Language.

library
In Axiom, a collection of compiled modules respresenting *category* or *domain* constructors.

lineage

the sequence of default packages for a given domain to be searched during *dynamic lookup*. This sequence is computed first by ordering the category ancestors of the domain according to their *level number*, an integer equal to the minimum distance of the domain from the category. Parents have level 1, parents of parents have level 2, and so on. Among categories with equal level numbers, ones that appear in the left-most branches of `Join`s in the source code come first. See a **Cross Reference** page of a constructor in Browse. See also *dynamic lookup*.

LISP

acronym for List Processing Language, a language designed for the manipulation of non-numerical data. The Axiom library is translated into LISP then compiled into machine code by an underlying LISP system.

list

an object of a **List** domain.

literal

an object with a special syntax in the language. In Axiom, there are five types of literals: booleans, integers, floats, strings, and symbols.

local

(syntax) A keyword used in user-defined functions to declare that a variable is a *local variable* of that function. Because of default assumptions on variables, such a declaration is often not necessary but is available to the user for clarity when appropriate.

local variable

(of a function) a variable bound by that function and such that its binding is invisible to any function that function calls. Also called a *lexical* variable. By default in the interpreter:

1. any variable x that appears on the left-hand side of an assignment is normally regarded a local variable of that function. If the intention of an assignment is to change the value of a *global variable* x, the body of the function must then contain the statement $free\,x$.

2. any other variable is regarded as a *free variable*.

An optional declaration $local\,x$ is available to declare explicitly a variable to be a local variable. All formal parameters are local variables to the function.

loop

1. an expression containing a `repeat`. 2. a collection expression having a `for` or a `while`, for example, $[f(i)\,for\,i\,in\,S]$.

loop body

the part of a loop following the `repeat` that tells what to do each iteration. For example, the body of the loop $for\,x\,in\,S\,repeat\,B$ is B. For a collection expression, the body of the loop precedes the initial `for` or `while`.

macro

1. *(interactive syntax)* An expression of the form $macro\ a\ ==\ b$ where a is a *symbol* causes a to be textually replaced by the expression b at *parse* time. 2. An expression of the form $macro\ f(a)\ ==\ b$ defines a parameterized macro expansion for a parameterized form f. This macro causes a form $f(x)$ to be textually replaced by the expression c at parse time, where c is the expression obtained by replacing a by x everywhere in b. See also *definition* where a similar substitution is done during *evaluation*. 3. *(programming language syntax)* An expression of the form $a\ ==>\ b$ where a is a symbol.

mode

a type expression containing a question-mark (**?**). For example, the mode POLY ? designates *the class of all polynomials over an arbitrary ring.*

mutable

objects that contain pointers to other objects and that have operations defined on them that alter these pointers. Contrast *immutable*. Axiom uses *pointer semantics* as does *LISP* in contrast with many other languages such as PASCAL that use *copying semantics*. See *pointer semantics* for details.

name

1. a *symbol* denoting a *variable*, such as the variable x. 2. a *symbol* denoting an *operation*, that is, the operation $divide : (Integer, Integer) - > Integer$.

nullary

a function with no arguments, for example, **characteristic**; operation or function with *arity* zero.

object

a data entity created or manipulated by programs. Elements of domains, functions, and domains themselves are objects. The most basic objects are literals; all other objects must be created by functions. Objects can refer to other objects using pointers and can be *mutable*.

object code

code that can be directly executed by hardware; also known as *machine language*.

operand

an argument of an *operator* (regarding an operator as a *function*).

operation

an abstraction of a *function*, described by a *signature*. For example, $fact : NonNegativeInteger - > NonNegativeInteger$ describes an operation for "the factorial of a (non-negative) integer."

operator

special reserved words in the language such as $+$ and $*$; operators can be either *prefix* or *infix* and have a relative *precedence*.

overloading

the use of the same name to denote distinct operations; an operation is identified

by a *signature* identifying its name, the number and types of its arguments, and its return types. If two functions can have identical signatures, a *package call* must be made to distinguish the two.

package
a special case of a domain, one for which the exported operations depend solely on the parameters and other explicit domains (contain no $). Intuitively, packages are collections of (*polymorphic*) functions. Facilities for integration, differential equations, solution of linear or polynomial equations, and group theory are provided by packages.

package call
(syntax) an expression of the form $e\$P$ where e is an *application* and P denotes some *package* (or *domain*).

package constructor
same as *domain constructor*.

parameter
see *argument*.

parameterized datatype
a domain that is built on another, for example, polynomials with integer coefficients.

parameterized form
a expression of the form $f(x, y)$, an *application* of a function.

parent
(of a domain or category) a category which is explicitly declared in the source code definition for the domain either to the left of the `with` or as an *export* of the domain. See *category extension*. See also a **Cross Reference** page of a constructor in Browse.

parse
1. (verb) to transform a user input string representing a valid Axiom expression into an internal representation as a tree-structure; the resulting internal representation is then "interpreted" by Axiom to perform some indicated action.

partially ordered set
a set with a reflexive, transitive and antisymetric *binary* operation.

pattern matching
1. (on expressions) Given an expression called the "subject" u, the attempt to rewrite u using a set of "rewrite rules." Each rule has the form $A == B$ where A indicates an expression called a "pattern" and B denotes a "replacement." The meaning of this rule is "replace A by B." If a given pattern A matches a subexpression of u, that subexpression is replaced by B. Once rewritten, pattern matching continues until no further changes occur. 2. (on strings) the attempt to match a string indicating a "pattern" to another string called a "subject", for example, for the purpose of identifying a list of names. In Browse, users may enter search strings for the purpose of identifying constructors, operations, and

attributes.

pile
alternate syntax for a block, using indentation and column alignment (see also *block*).

pointer
a reference implemented by a link directed from one object to another in the computer memory. An object is said to *refer* to another if it has a pointer to that other object. Objects can also refer to themselves (cyclic references are legal). Also more than one object can refer to the same object. See also *pointer semantics*.

pointer semantics
the programming language semantics used in languages such as LISP that allow objects to be *mutable*. Consider the following sequence of Axiom statements:
$x : VectorInteger := [1, 4, 7]$
$y := x$
$swap!(x, 2, 3)$
The function **swap** is used to interchange the second and third value in the list x, producing the value $[1, 7, 4]$. What value does y have after evaluation of the third statement? The answer is different in Axiom than it is in a language with *copying semantics*. In Axiom, first the vector $[1, 2, 3]$ is created and the variable x set to point to this object. Let's call this object V. Next, the variable y is made to point to V just as x does. Now the third statement interchanges the last 2 elements of V (the ! at the end of the name **swap** tells you that this operation is destructive, that is, it changes the elements *in place*). Both x and y perceive this change to V. Thus both x and y then have the value $[1, 7, 4]$. In PASCAL, the second statement causes a copy of V to be stored under y. Thus the change to V made by the third statement does not affect y.

polymorphic
a *function* (for example, one implementing an *algorithm*) defined with categorical types so as to be applicable over a variety of domains (the domains which are members of the categorical types). Every Axiom function defined in a domain or package constructor with a domain-valued parameter is polymorphic. For example, the same matrix + function is used to add "matrices over integers" as "matrices over matrices over integers."

postfix
an *operator* that follows its single *operand*. Postfix operators are not available in Axiom.

precedence
(syntax) refers to the so-called *binding power* of an operator. For example, $*$ has higher binding power than $+$ so that the expression $a + b * c$ is equivalent to $a + (b * c)$.

precision
the number of digits in the specification of a number. The operation **digits** sets this for objects of **Float**.

predicate
1. a Boolean-valued function, for example, $odd : Integer- > Boolean$. 2. a Boolean-valued expression.

prefix
(syntax) an *operator* such as $-$ that is written *before* its single *operand*. Every function of one argument can be used as a prefix operator. For example, all of the following have equivalent meaning in Axiom: $f(x)$, fx, and $f.x$. See also *dot notation*.

quote
the prefix *operator* ' meaning *do not evaluate*.

Record
(basic domain constructor) a domain constructor used to create an inhomogeneous aggregate composed of pairs of selectors and values. A **Record** domain is written in the form $Record(a1 : D1, \ldots, an : Dn)$ $(n > 0)$ where $a1, \ldots, an$ are identifiers called the *selectors* of the record, and $D1, \ldots, Dn$ are domains indicating the type of the component stored under selector an.

recurrence relation
A relation that can be expressed as a function f with some argument n which depends on the value of f at k previous values. In most cases, Axiom will rewrite a recurrence relation on compilation so as to *cache* its previous k values and therefore make the computation significantly more efficient.

recursion
use of a self-reference within the body of a function. Indirect recursion is when a function uses a function below it in the call chain.

recursive

1. A function that calls itself, either directly or indirectly through another function. 2. self-referential. See also *recursive*.

reference
see *pointer*

relative
(of a domain) A package that exports operations relating to the domain, in addition to those exported by the domain. See a **Cross Reference** page of a constructor in Browse.

representation
a *domain* providing a data structure for elements of a domain, generally denoted by the special identifier *Rep* in the Axiom programming language. As domains are abstract datatypes, this representation is not available to users of the domain, only to functions defined in the *function body* for a domain constructor. Any domain can be used as a representation.

reserved word
a special sequence of non-blank characters with special meaning in the Axiom language. Examples of reserved words are names such as `for`, *if*, and *free*, operator names such as + and **mod**, special character strings such as `==` and `:=`.

retraction
to move an object in a parameterized domain back to the underlying domain, for example to move the object 7 from a "fraction of integers" (domain **Fraction Integer**) to "the integers" (domain **Integer**).

return
when leaving a function, the value of the expression following `return` becomes the value of the function.

ring
a set with a commutative addition, associative multiplication, a unit element, where multiplication is distributive over addition and subtraction.

rule
(syntax) 1. An expression of the form $ruleA == B$ indicating a "rewrite rule." 2. An expression of the form $rule(R1;...;Rn)$ indicating a set of "rewrite rules" $R1,...,Rn$. See *pattern matching* for details.

run-time
the time when computation is done. Contrast with *compile-time*, and *dynamic* as opposed to *static*. For example, the decision of the intepreter to build a structure such as "matrices with power series entries" in response to user input is made at run-time.

run-time check
an error-checking that can be done only when the program receives user input; for example, confirming that a value is in the proper range for a computation.

search string

a string entered into an *input area* on a HyperDoc screen.

selector
an identifier used to address a component value of a **Record** datatype.

semantics
the relationships between symbols and their meanings. The rules for obtaining the *meaning* of any syntactically valid expression.

semigroup
(algebra) a *monoid* which need not have an identity; it is closed and associative.

side effect
action that changes a component or structure of a value. See *destructive operation* for details.

signature
(syntax) an expression describing the type of an *operation*. A signature has the form *name* : *source−* > *target*, where *source* is the type of the arguments of the operation, and *target* is the type of the result.

small float
an object of the domain **DoubleFloat** for floating-point arithmetic as provided by the computer hardware.

small integer
an object of the domain **SingleInteger** for integer arithmetic as provided by the computer hardware.

source
the *type* of the argument of a *function*; the type expression before the − > in a *signature*. For example, the source of $f : (Integer, Integer)− > Integer$ is $(Integer, Integer)$.

sparse
data structure whose elements are mostly identical (a sparse matrix is one filled mostly with zeroes).

static
that computation done before run-time, such as compilation. Contrast *dynamic*.

step number
the number that precedes user input lines in an interactive session; the output of user results is also labeled by this number.

stream
an object of **Stream(R)**, a generalization of a *list* to allow an infinite number of elements. Elements of a stream are computed "on demand." Streams are used to implement various forms of power series.

string
an object of domain **String**. Strings are literals consisting of an arbitrary sequence of characters surrounded by double-quotes ("), for example,

"*Lookhere!*".

subdomain
(basic concept) a *domain* together with a *predicate* characterizing the members of the domain that belong to the subdomain. The exports of a subdomain are usually distinct from the domain itself. A fundamental assumption however is that values in the subdomain are automatically coerceable to values in the domain. For example, if n and m are declared to be members of a subdomain of the integers, then *any binary* operation from **Integer** is available on n and m. On the other hand, if the result of that operation is to be assigned to, say, k, also declared to be of that subdomain, a *run-time* check is generally necessary to ensure that the result belongs to the subdomain.

such that clause
(syntax) the use of | followed by an expression to filter an iteration.

suffix
(syntax) an *operator* that is placed after its operand. Suffix operators are not allowed in the Axiom language.

symbol
objects denoted by *identifier* literals; an element of domain **Symbol**. The interpreter, by default, converts the symbol x into **Variable(x)**.

syntax
rules of grammar and punctuation for forming correct expressions.

system commands
top-level Axiom statements that begin with). System commands allow users to query the database, read files, trace functions, and so on.

tag
an identifier used to discriminate a branch of a **Union** type.

target
the *type* of the result of a *function*; the type expression following the -> in a *signature*.

top-level
refers to direct user interactions with the Axiom interpreter.

totally ordered set
(algebra) a partially ordered set where any two elements are comparable.

trace
use of system function)trace to track the arguments passed to a function and the values returned.

tuple
an expression of two or more other expressions separated by commas, for example, $4, 7, 11$. Tuples are also used for multiple arguments both for applications (for example, $f(x, y)$) and in signatures (for example, $(Integer, Integer)->$ $Integer$). A tuple is not a data structure, rather a syntax mechanism for group-

ing expressions.

type

The type of any *category* is the unique symbol *Category*. The type of a *domain* is any *category* to which the domain belongs. The type of any other object is either the (unique) domain to which the object belongs or a *subdomain* of that domain. The type of objects is in general not unique.

Type

a category with no operations or attributes, of which all other categories in Axiom are extensions.

type checking

a system function that determines whether the datatype of an object is appropriate for a given operation.

type constructor

a *domain constructor* or *category constructor*.

type inference

when the interpreter chooses the type for an object based on context. For example, if the user interactively issues the definition $f(x) == (x + \%i) * *2$ then issues $f(2)$, the interpreter will infer the type of f to be $Integer- >$ $ComplexInteger$.

unary

operation or function with *arity* 1.

underlying domain

for a *domain* that has a single domain-valued parameter, the *underlying domain* refers to that parameter. For example, the domain "matrices of integers" (**Matrix Integer**) has underlying domain **Integer**.

Union

(basic domain constructor) a domain constructor used to combine any set of domains into a single domain. A **Union** domain is written in the form $Union(a1 : D1, ..., an : Dn)$ $(n > 0)$ where $a1$, ..., an are identifiers called the *tags* of the union, and $D1$, ..., Dn are domains called the *branches* of the union. The tags

$$ai$$

are optional, but required when two of the

$$Di$$

are equal, for example, $Union(inches : Integer, centimeters : Integer)$. In the interpreter, values of union domains are automatically coerced to values in the branches and vice-versa as appropriate. See also *case*.

unit

(algebra) an invertible element.

user function

a function defined by a user during an interactive session. Contrast *built-in function*.

user variable

a variable created by the user at top-level during an interactive session.

value

1. the result of evaluating an expression. 2. a property associated with a *variable* in a *binding* in an *environment*.

variable

a means of referring to an object, but not an object itself. A variable has a name and an associated *binding* created by *evaluation* of Axiom expressions such as declarations, assignments, and definitions. In the top-level *environment* of the interpreter, variables are global variables. Such variables can be freely referenced in user-defined functions although a *free* declaration is needed to assign values to them. See *local variable* for details.

Void

the type given when the *value* and *type* of an expression are not needed. Also used when there is no guarantee at run-time that a value and predictable mode will result.

wild card

a symbol that matches any substring including the empty string; for example, the search string "`*an*`" matches any word containing the consecutive letters "`a`" and "`n`".

workspace

an interactive record of the user input and output held in an interactive history file. Each user input and corresponding output expression in the workspace has a corresponding *step number*. The current output expression in the workspace is referred to as %. The output expression associated with step number n is referred to by %%(n). The k-th previous output expression relative to the current step number n is referred to by %%($-k$). Each interactive *frame* has its own workspace.

Appendix H

License

PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Testing involves function calls which are intended to fail in
the normal course of testing. In order to distinguish cases
where things fail by design you can issue the command
{\tt)set message test on}. This inhibits the message
"Daly Bug". You can now distinguish between intentional
failures which do not contain the "Daly Bug" message and
unintentional failures which do.

RDJ note to RSS: Expressions not statements or lines-- By an
expression I mean any syntactically correct program fragment.
Everything in AXIOM is an expression since every fragment has a value
and a type. In most languages including LISP, a "statement" is
different from an expression: it is executed for side-effect only and
an error is incurred if you assign it a value. This "gimmick" takes
care of incomplete expressions such as "if x > 0 then y" in blocks.
In LISP, "u := (if x > 0 then y)" is illegal but in AXIOM it is legal.
Also, in AXIOM the value of a repeat loop is void even though you
might be be able to prove that it always returns a valid value (you
have an example of this)! This will be considered a bug not a
feature. But it is how things stand. In any case---this point should
be in a box somewhere since it is key to a user's understanding to the
language. I am not sure where. You only gain an appreciation for it
after awhile in chapter 5.

Bibliography

[1] Lamport, Leslie, *LaTeX: A Document Preparation System,*
Reading, Massachusetts, Addison-Wesley Publishing Company, Inc., 1986.
ISBN 0-201-15790-X

[2] Knuth, Donald, *The T$_E$Xbook*
Reading, Massachusetts, Addison-Wesley Publishing Company, Inc., 1984.
ISBN 0-201-13448-9

[3] Jenks, Richard D. and Sutor, Robert S.,
Axiom, The Scientific Computation System
Springer-Verlag, New York, NY 1992 ISBN 0-387-97855-0