NewAutodoc

William Sit

Original by Tim Daly
New Documentation August 13, 2006
Last updated August 14, 2006

Abstract

This is an example of a pamphlet file, which is the source file format used to document all the code in the Axiom system at every level. This example illustrates the use of a lisp function that reads in a particular Axiom pamphlet file, looks for lines that define Axiom constructors in the form of a beginning chunk line, and returns a list of abbreviations and a list of corresponding long names for the constructors included in the file. This version is based on Tim Daly's version [1], with major revisions in the documentation.

Contents

1 Revision Notes

This is a revised version of Tim Daly's autodoc.lisp.pamphlet (AutoDocPamphlet, [1]). It is also my first attempt at editing a pamphlet. I am not lisp-literate and I find Tim's version rather confusing at first in the way he documents the lisp code. I don't know if the code chunks, which I rearranged totally (so a diff would be pointless) would still work. I am using this page to see if the code after a tangle will still be the same. It is not my intention to change (or test) any actual code. This is an exercise mainly in documentation only, not lisp code revision. There are a few lines of lisp code that I could not guess the exact syntax. Any corrections of mistakes I made and comments would be welcome.

Besides the order by which the lisp code chunks are documented, I also insist that the parentheses be balanced in any code chunk, which is not the case in Tim's version. Without balanced parentheses, the outermost structure of the code is obscured. In other words, a code chunk should be a complete idea, not a partial one with dangling clauses (since it seems impossible to *continue* an incomplete code chunk without starting a new chunk name). If I am not mistaken, there was one extra closing parenthesiss in Tim's code at the end. I don't think this is any production code anyway, but if it were, and if I am correct, insisting the external and internal balance of parentheses would minimize this type of errors. I also modified two substraction code Tim wrote where he used infix notation, to prefix notation. The input file to the program should also be closed when done.

This exercise illustrates that revising even just the documentation portion of a pamphlet file may accidentally changed the code due to different ways to place parentheses in the code chunks. Even for this very simple lisp function, a lot of effort is used to make sure the tangled code remains unchanged (except where it is intentional). I cannot be certain that revision of pamphlets, even just for documentation, is safe without a direct comparison of the extracted codes and even code testing.

2 To Do List

Except for minor changes as indicated, the lisp code remains the same as in Tim's version. However, I have not yet tested the code and there are lines where I am not following and so additional documentation seems necessary for lisp-illiterates like me. The code need to be tested, and as suggested by Ralf Hemmecke, when it is successfully run against an input file, the actual output should be given in this pamphlet. Perhaps there should even be a section on expected output, and if future modification of the code produces variations in output, they may be detected automatically. Please feel free (BE BOLD) to add your documentation and revise as you see fit.

3 Usage::srcabbrevs

The actual lisp executable for the program calls the lisp function `srcabbrevs` which takes the full path name of a source file `sourcefile` as its only parameter. Here, the sample file is `acplot.spad.pamplet` from the directory `/new/patch38/src/algebra/`.

⟨*usage*⟩≡
```
(srcabbrevs "/new/patch38/src/algebra/acplot.spad.pamphlet")
```

4 Function `srcabbrevs`

By design the beginning chunk line for an Axiom constructor consists of a chunk name followed by an "=" sign. The chunk name is enclosed in a double pair of angular brackets << ... >>, where the ... has three fields. The first field is an identification for the constructor class, which may be one of {category, domain, package}. The second field is the abbreviation for the constructor. The third field is the full name of the constructor (without its parameters). The three fields are separated by space or spaces. The fields themselves must not have any embedded space characters.

The lisp function `srcabbrevs` takes the input `sourcefile` and opens it.

⟨*open the source file*⟩≡
```
with-open-file (in sourcefile)
```

It then reads each line and stores it in the variable `expr`. The variable `expr` is set to the string "done" if end of file is reached. For each line before end of file, it trims off trailing spaces and stores the trimmed line again in `expr`. If it reaches end of file, it throws an error flag `done`.

⟨*for each line do*⟩≡
```
loop
(setq expr (read-line in nil 'done))
(when (eq expr 'done) (throw 'done nil))
(setq expr (string-trim '(#\space) expr))
```

The parser ignores any line that does not begin with the beginning code chunk characters << and ignores any line if the code chunck line does not define an Axiom constructor. If it is a constructor, it looks for lines beginning with a chunk name indicating the constructor class and ending with an equal sign. The code allows for shortened constructor class: "pa" for "package", "do" for "domain", and "ca" for "category". This conditional is defined below.

⟨*when a chunk name is a constructor definition*⟩≡
```
when
(and (> (length expr) 4)
(or
(string= "<<pa" (subseq expr 0 4))
(string= "<<do" (subseq expr 0 4))
(string= "<<ca" (subseq expr 0 4)))
(char= (elt expr (- (length expr) 1)) #\=))
```

If this condition is satisfied, it removes the three characters >>= that ends the chunk line in `expr`.

⟨*remove the trailing chunk characters*⟩≡
```
(setq expr (subseq expr 0 (- (length expr) 3)))
```

It then extracts the third field from the string which is the long name by locating the position of the last space character in `expr` and saves the position in a variable `point`.

⟨*capture the long name*⟩≡
```
(setq point (position #\space expr :from-end t :test #'char=))
(format t "~a ~a ~a~%" expr point longnames)
```

It places the full name of the constructor into the list `longnames` using a special format. (I do not understand the syntax of the format line).

To extract the second field from the string which is the abbreviation, it trims extra spaces after the abbreviation, marks the position of the first space before the abbreviation, trims the extra space and pushes the abbreviation in the list `names`.

⟨*capture the abbreviation*⟩≡
```
(setq mark
(position #\space
(string-right-trim '(#\space)
(subseq expr 0 (- point 1))) :from-end t))
(push (string-trim '(#\space) (subseq expr mark point)) names)
```

The two lists are then returned.

⟨*output*⟩≡
```
(values names longnames)
```

The error flag `done` is caught here:

⟨*until done*⟩≡
```
catch 'done
```

It seems that there should be some code to close the file when end of file is reached. Putting the code chunks together defines the function `srcabbrevs`. I do not follow the first line in the definition: (let (in expr start end names longnames) . . . .

⟨*srcabbrevs*⟩≡
```
  (defun srcabbrevs (sourcefile)
  (let (in expr start end names longnames)
  (⟨until done⟩
  (⟨open the source file⟩
  (⟨for each line do⟩
  (⟨when a chunk name is a constructor definition⟩
  ⟨remove the trailing chunk characters⟩
  ⟨capture the long name⟩
  ⟨capture the abbreviation⟩
  )))))
  ⟨output⟩))
```

5 Main Program and Output

This defines the complete program `autodoc`.

⟨*autodoc*⟩≡
```
  ⟨srcabbrevs⟩
  ⟨usage⟩
```

The expected output when this program is successfully run under lisp is not known at this time since the code has not been tested.

6 Makefile

The makefile specifies the way to extract the documentation and code and run it from this pamphlet file. The first three lines defines environment variables `TANGLE, WEAVE, LISP` giving the locations the executables and options. After changing directory to where this pamphlet file is stored (under the filename autodoc.lisp.pamphlet), `make code` produces a file autodoc.lisp containing the resulting lisp program, `make doc` produces a latex source autodoc.tex and a dvi file autodoc.dvi, and finally, `make run` runs the lisp program (I think `make run` should be conditioned on the existence of autodoc.lisp). I do not know what `make remake` is supposed to do. My guess is it will generate Makefile. The terminal command `make all` will create the lisp program, documentation, and run it.

⟨*⟩≡

```
TANGLE=/usr/local/bin/notangle -t8
WEAVE=/usr/local/bin/noweave -delay
LISP=/new/patch38/obj/linux/bin/lisp

all: code doc run

code: autodoc.lisp.pamphlet
@${TANGLE} -Rautodoc autodoc.lisp.pamphlet >autodoc.lisp

doc: autodoc.lisp.pamphlet
@${WEAVE} autodoc.lisp.pamphlet >autodoc.tex
@latex autodoc.tex
@latex autodoc.tex

run: autodoc.lisp
@cat autodoc.lisp | ${LISP}

remake:
@${TANGLE} autodoc.lisp.pamphlet >Makefile
```

References

[1] Tim Daly, AutoDoc, autodoc.lisp.pamphlet