`CL-WEB` v0.5 - Literate Programming Tools Implemented in ANSI Common Lisp

Clifford Yapp, Waldek Hebisch, Kai Kaminski

February 21, 2023

Abstract

NOWEB is the classic language agnostic tool used for literate programming in the style of Knuth's original WEB. This tool implements a method in lisp for extracting the source code from literate documents.

Contents

1 Updates

`(2/15/07)` - Initial v0.1 released
`(2/16/07)` - Walter Hebisch provides linear time read-in-chunk
`(2/17/07)` - Minor change to check for @ character, copyright fix
`(4/29/07)` - v0.4 - implement finite state version per Waldek Hebisch's design. Switch to defstruct.
`(4/30/07)` - v0.5 - rename some code, add eval-whens, fix bug in reference handling, more text.

2 Copyright and License

This program is available under the Modified BSD license. The wording of the file was changed slightly, as there is no organization associated with the creation of this file, but if there is any question the license is intended to be Modified BSD.

⟨*copyright*⟩≡

⟨*license*⟩≡

3 Introduction - NOWEB and Literate Programming

Literate Programming describes a programming and development methodology that focuses not on source code or documentation but writing a *work of literature* containing both, integrated and organized in a manner that interweaves source code and the background material that puts it in context for a human reader.

Traditional programming approaches normally consist of writing and debugging source code intended for machine consumption, and maintaining a separate work intended to explain the system to its users and developers. This can create challenges in situations where someone other than the original author of the source code needs to modify it, as the documentation in most cases is only loosely related to the source code and may be out of date. In many cases even the original programmer, faced with the code he/she wrote months or years later, may be at a loss to explain design decisions and motivations for a particular piece of code. This phenomena, particularly in large or complex programs, can lead to a number of undesirable situations. The simplest one is increased time spend debugging old code where changes may produce unintended side effects. More serious problems arise as the complexity of the code increases - as change becomes increasingly difficult due to unknown/undocumented sections of code, work on new features can grind to a hault. Even if this does not happen, difficulty in understanding the existing code will limit the number of potential contributors to a project.

Literate Programming is one approach to avoiding (or at least minimizing) these issues. A literate program will describe the concepts, the key design issues, and other important information as if the author was writing a research paper or report. Indeed, the report IS being written, but in addition to the report the code implementing the ideas is included as well. This means that anyone with sufficient background to understand a paper on the subject should be able to puzzle out the program, as well.

The rise of Literate Programming[**?**, **?**] traces back to Donald Knuth, the author of the TeX typesetting system. In addition to being a pioneer of the concept itself, Knuth implemented WEB[**?**] to put the concept into practice.[1] The original WEB system was intended to be used with Pascal, and other implementations arose for other languages. The CWEB system by Knuth and Levy [**?**, **?**, **?**, **?**] adapted WEB to C and C++. Norman Ramsey realized that it would be possible to create an implementation of a literate programming system which was language agnostic, and in 1989 began developing NOWEB[**?**, **?**, **?**]. NOWEB's virtues are simplicity and language independence. NOWEB places less focus on "prettyprinting" code, which refers to the practice of typesetting code in such a fashion that key language features and program structure are emphasized. Other systems customized to a specific language can take into account the structure of the source code when typesetting - NOWEB supports this activity via filters, but does not implement it as a core internal feature.

These tools all convert between the original literate source format and formats understandable by either a typesetting system (usually TeX) or a compiler. Because the order of the source code in the original document may bear little or no resemblance to the order in which the machine needs to see it for evaluation, it must be extracted from the document. This is further complicated by the emphasis of literate programming on ideas rather than code dictating the order of a programmer's work. WEB documents thus are not only interwoven text and source code, but chunks of code may themselves contain references to other chunks of code in the document. There is usually a root chunk which places the individual pieces in the correct order, but each piece may in turn reference other pieces and all of them must be eventually put in the correct sequence to allow the machine to read them. This process is laborious and error prone if done by hand, so extracting the code in machine readable form is one of the primary jobs of a literate programming tool. In NOWEB, the specific command that does this job is called notangle.[2]

It cannot be denied that literate programming involves more work than simply writing the machine code, and it may entail more effort than machine code + a separate user manual. This form of programming demands a thorough understanding of the problem being solved. It does not lend itself to poorly considered "quick and dirty" solutions, as the minimum effort to create a literate program involves sufficient knowledge

[1] Although there are certain conceptual similarities to hyperlinked documents, Knuth's WEB is not a client or server for protocals used in the World Wide Web, which as of 2007 is the common interpertation of the word "web" in the context of computers.

[2] Knuth appears to regard the source code interwoven with text as the untangled form of the code, and the code ordered in such a way that the machine understands it as the "tangled" form. This is consistent with the document centric view of the program but can be confusing to programmers accustomed to writing stand alone code files - from the latter point of view the "tangled" form of the code is the one where the source is mixed up in the document.

of the problem to document the how and why of its solution. When writing programs intended to last for long periods of time and be maintained for long periods of time by many different hands, this is critically important. TeX is a good example - a system which solves its assigned problem so well it is used decades after its original development.

NOWEB implements a very general, flexible system. Axiom's use of a very well defined pamphlet structure means this project (`cl-web`) requires only a subset of NOWEB's abilities. The system should be able (with some work) to cover more general cases if needed - for now, the basics are enough.

4 `cl-web`'s Working Chunk Structure

In general, a chunk[3] may be viewed as having the following structure:

```
**********************************************
**********************************************
[1st delimiter]*chunk-name*[2nd delimiter]
----------------------------------------------
---[3rd delimiter]reference[4th delimiter]-----
----------------------------------------------
[5th delimiter]
**********************************************
**********************************************
```

The 1st, 2nd, and 5th delimiters occur only once per chunk, while the 3rd and 4th are present for each reference contained within the chunk content. This version of WEB will require the 5th delimiter start in the first column of the text file (a.k.a immediately after a newline).

5 Initial Definitions

The first step is to define a common lisp package in which this functionality will reside, and switch into that package to define subsequent code. In this case the package name `cl-web` will be used. At this time Lisp also needs to know which functions and variables are intended to be visible outside the package. The external command here is the tangle[4] command, so we identify it in the export list. A utility variable containing the `cl-web` package name is defined here for later use. `eval-when` makes sure it is defined at both the time of macro expansion as well as the more normal points in the loading process. This allows a compile command to correctly expand the various macros when compiling the file.

⟨*package-def*⟩≡
```
(defpackage cl-web
  (:use :cl)
  (:export #:tangle))

(in-package :cl-web)

(eval-when (:compile-toplevel :load-toplevel :execute) (defvar webpkg :cl-web))
```

[3]A *chunk* is a region of the literate document containing source code, surrounded by markup which identifies it to the system.

[4]NOWEB's use of the "no" prefix was (according to the NOWEB FAQ) partially motivated by the proliferation of language specific versions of WEB - hence "no" web was a WEB for no particular language. Rather than adopt that convention, we will simply use "tangle" to identify the command that extracts the source from the document.

File reading, hashing, and other techniques used to implement the `tangle` operation require either arrays of data or strings to achieve maximum speed - reading and scanning the file are best done in an array of bytes but hashing chunk names works best with strings. As a result, conversion from an array data structure to a string data structure is needed. `array-to-string`, like several other functions, also contains an instruction to the compiler to optimize the compilation of this function for speed. The type statement used for `ar` identifies to the compiler the exact type of the incoming array and allows for optimizations at compile time. Once these facts are declared, a let statement defines a local variable for the length of the array and makes a string of that same length. From that point, an interation over the array copies the contents character by character into the string, and the string is returned.

⟨*array-to-string*⟩≡
```
(defun array-to-string (ar)
    (declare (optimize (speed 3))
             (type (simple-array (unsigned-byte 8) (*)) ar))
    (let* ((len (array-dimension ar 0))
           (str (make-string len)))
          (dotimes (i len)
               (setf (aref str i) (code-char (aref ar i))))
          str))
```

When reading in chunks and storing them, a hash table is used. It is initialized at the beginning, since it is used throughout the program. An important feature to note here is that this hash table uses the "equal" function for equality testing, which is necessary for the hash table to work with string keys. Chunk names are strings, and a default hash table configuration in Lisp will not accept them.

⟨*hashtable*⟩≡
```
(defparameter *chunk-hash-table* (make-hash-table :test 'equal))
```

There are two constants we must also define - newline-code and space-code - that will be common to all pamphlet files handled by `cl-web`:

⟨*first-constants*⟩≡
```
(defconstant newline-code 10)
(defconstant space-code (char-code #\ ))
```

6 Delimiters - Recognizing Regions

When parsing the input file, the features to identify are

- The start of a chunk, which is also the start of the region containing the name of the chunk.
- The end of the name of the chunk, which will also identify the start of the body of the chunk.
- The start of a reference marker containing a reference to another chunk.
- The end of a reference marker containing a reference to another chunk.
- The end of a chunk body, which also constitutes the end of the chunk.

These variables also need to be available at macro expansion time.

⟨*tangle-delimiter-variables*⟩≡
```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (progn
    (defvar chunk-start-delimiter "<<")
    (defvar chunk-name-end-delimiter ">>=")
    (defvar chunk-ref-start-delimiter "<<")
    (defvar chunk-ref-end-delimiter ">>")
    (defvar chunk-end-delimiter "@")))
```

In order for an efficient parsing of the file to be carried out, these delimiters must all be considered on a character by character basic. However, unless the strings denoting these delimiters are hard coded from the beginning, defconstant statements cannot be written in advance. To preserve flexibility, a Lisp macro is defined which will take as arguments the delimiter strings to be used for a particular operation and generate the definition code needed for each character.[5]

⟨*generate-marker-constants*⟩≡
```
(defmacro generate-marker-constants ()
  `(progn
    ,@(loop for i from 1
            for char across chunk-start-delimiter collect
            `(defconstant ,(intern (format nil "START-CHUNK-CHAR-~A" i))
              (char-code ,char)))
    ,@(loop for i from 1
            for char across chunk-name-end-delimiter collect
            `(defconstant ,(intern (format nil "END-NAME-CHAR-~A" i))
              (char-code ,char)))
    ,@(loop for i from 1
            for char across chunk-ref-start-delimiter collect
            `(defconstant ,(intern (format nil "START-REF-CHAR-~A" i))
              (char-code ,char)))
    ,@(loop for i from 1
            for char across chunk-ref-end-delimiter collect
            `(defconstant ,(intern (format nil "END-REF-CHAR-~A" i))
              (char-code ,char)))
    ,@(loop for i from 1
            for char across chunk-end-delimiter collect
            `(defconstant ,(intern (format nil "END-CHUNK-CHAR-~A" i))
              (char-code ,char)))))
```

[5]Thanks to luis on #lisp for patiently walking me through how to implement this macro. CY

Now that we have defined our delimiters and the method to handle generating constants from them, we call the macro to generate the actual code to be run:

⟨*make-axiom-constants*⟩≡
```
  (generate-marker-constants)
```

7 Chunk as a Lisp Structure

Because the key information defining a chunk will need to be stored in a hash table, a structure to hold that information is needed. In this case, a defstruct is defined to hold the original line number in the pamphlet file in which the chunk is defined, and a list holding the positions corresponding to key chunk structural points. The initial values are set to zero and an empty list. The defstruct command will automatically create calls `chunk-contents` and `chunk-line-num` to access the slots in the structure - these will be used to alter the content of a chunk object. Additonally, `make-chunk` will create a new instance of this structure.[6]

⟨*chunk-structure*⟩≡
```
  (defstruct chunk
    (line-num 0)
    (contents '()))
```

The `contents` slot is where the actual structure of the chunk is stored. The format used is:

(((num1 num2) “name1” (num3 num4) “name2” (num5 num6)...))

where the numbers represent positions in the array containing the file and names represent included chunks.

7.1 New Chunks and Adding Content to Chunks

Chunk creation and modification is all handled within a single function, `add-to-chunk-contents`. In order to change the contents of a chunk, the first thing to do is check if the chunk exists. If it doesn't yet exist, it is created and both contents and the line number are added. If it does exist, the addtional content is added.

⟨*add-to-chunk-contents*⟩≡
```
  (defun add-to-chunk-contents (name content &optional (linenum))
      (declare (optimize (speed 3)))
      (let ((cur-chunk (gethash name *chunk-hash-table*)))
           (if (not cur-chunk)
               (progn
                 (setf cur-chunk (setf (gethash name *chunk-hash-table*)
                                       (make-chunk)))
                 (push content (chunk-contents (gethash name *chunk-hash-table*)))
                 (setf (chunk-line-num (gethash name *chunk-hash-table*)) linenum))
               (push content (chunk-contents (gethash name *chunk-hash-table*))))))
```

[6]As of v0.5 the line number feature is not yet active.

7.2 Reading Pamphlets into Lisp

Transferring a file from its storage location on the hard disk or operating system memory into Lisp is actually fairly important for speed - the right techniques must be used when the fastest possible read operation is needed.

First, the function `file-to-array` creates an array with a size equal to the file it will read, and then uses read-sequence to do a fast import of the file into the array. The newline-code defined earlier is assigned to the end of the array and the resulting array is returned as the result of the function call.

⟨*file-to-array*⟩≡
```
(defun file-to-array (f)
    (let* ((b-len (file-length f))
           (buff (make-array (list (+ b-len 1))
                             :element-type '(unsigned-byte  8))))
        (read-sequence buff f)
        (setf (aref buff b-len) newline-code)
        buff))
```

Now that we have a fast function for importing the file, we call it using a function which opens the file for reading. Note the flags defining the type of data the open file command assumes - these are important to the fast read operation as they match the array element type used in `file-to-array`.

⟨*read-pamphlet-file*⟩≡
```
(defun read-pamphlet-file (name)
    (with-open-file (f name :element-type '(unsigned-byte  8))
         (file-to-array f)))
```

8 Parsing the Pamphlet File - Locating Chunks and Chunk References

Now we arrive at the heart of the `tangle` operation - the actual identification of chunk definitions and references to other chunks within those definitions. There are a variety of techniques that can be used to perform this operation, but because of the potential complexity of the chunk relations and the importance of speed when dealing with large scale literate documents, the technique of finite automation will be used to scan for chunks.

8.1 Tangle as a Finite State Machine

Finite automation is basically a careful consideration of every point of the scanning process, and defining exactly what steps should happen in each situation. Finite automation is used for well defined tasks that can be specified completely, and because the `tangle` operation is well specified such a list of operations can be defined.

For the sake of simplicity, several assumptions are made in this implementation that may not be made in other implementations. The first assumption is that all chunk beginning and end tags appear in the first column of the text file, or from the parser viewpoint immediately after a newline character. The second assumption is no chunk names will contain a newline character, or in other words there are no multi-line chunk names.

There are 17 states which may occur during the processing of a pamphlet file:

Diagram Label	Description
Ss	normal start
Sn	normal
Scst	in chunk start tag
Scn	in chunk name
Scne	in end of name tag
Scnet	clearing trailing spaces after chunk name
Scb	in chunk body
Scn	new chunk line
Scr	in chunk reference start tag
Scrn	in chunk reference name
Scrne	in end of reference tag
Scrf	final ref char found
Scei	Inside end of chunk tag
Sceit	chars after end of chunk tag
Eub	Error unexpected buffer end
Egn	Error garbage after name
Eof	End of File

There is a slight abstraction in the description of states - a true diagram would have a state for each character in a given tag name, as that is the actual requirement for parsing. However, until the contents of the tags are know, a final diagram is not possible. Hence the use of the "in tag" states - these represent a "metastate" of all character states between the first and last character of a tag. The actions taken in the interior of the tags are constant except for the individual character against which the read character is compared, therefore this abstraction is safe. This limitation is also present in the code - as with the constant statements above, a macro must be used to generate the final scanning function once the characters of the tags have been defined.

There are also a variety of transition conditions that may be encountered, again using the previously mentioned abstraction of the tag interiors. In this case, to make the variety of labels more managable, only the read actions corresponding to the start of various tag types are uniquely labeled. Other situations use the abstractions "next character" and "final character" to describe the transitions, although the details of what character represents the transition will change depending on the state.

Diagram Label	Description
Nc	Normal character read (Any character other than those special in a given `tangle` context.)
Nl	Newline character read
Ef	End of file read
Scc	Read character corresponding to the first character of the start of chunk tag.
Scen	Read character corresponding to the first character of the start of chunk tag.
Scr	Read character corresponding to the first character of the start of chunk reference tag.
Scre	Read character corresponding to the first character of the end of chunk reference tag.
Sec	Read character corresponding to the first character in the end of chunk body tag.
Ntc	Read character corresponding to the next character of the current tag.
Ntcf	Final tag character read.
Ns	Space character read

With those definitions made, we can create the finite state machine diagram for the `tangle` process1. (A double circle is referred to as an accept state.)

Figure 1: Finite State Diagram of the TANGLE process.

The diagram is the map, but it is insufficient to specify all of the features required. The scan function must record the position of chunks as they are located in the string.

8.2 Automatic State Generation Based on Delimiter Strings

In order to handle arbitrary delimiter strings, the state conditions pertaining to the details of the delimiter structures must be auto-generated. There are five types of delimiters, each (potentially) with a beginning, middle, and end to them. There are three cases which must be individually delt with for each case - the start of a delimiter, the middle (for length ¿ 2), and the end (for length ¿ 1). These macros will refer to tags not yet in existence - they are defined later when the general scan function structure is written. All of these functions must be definied when the macro writing the scan function is run, hence the `eval-when` wrapped around each one.

8.2.1 1st - the Start of a Chunk Name

The first character of the first delimiter has two possible actions depending on delimiter string length. If length is exactly one, the recoginition of the single character completes the recognition of the delimiter and the state proceeds to the reading of the chunk name. Otherwise, the next character in the delimiter is checked for. The supplied variable (i) should be the string length of the delimiter in question. Because this first recognition occurs inside another state and is not a complete state definition in and of itself, it does not generate its own state name.

⟨*1ststart*⟩≡
```
(eval-when (:compile-toplevel :load-toplevel)
(defun expand1ststart (i)
  (if (= 1 i)
      ‘(if (eql code start-chunk-char-1)
        (progn
          (setf start-pos (+ pos 1))
```

```
            (go in-chunk-name)))
        `(if (eql code start-chunk-char-1)
          (go chunk-start-tag-1))))
  )
```

If delimiter string length is greater than one, there will be an end character. Here, the tag name must be generated as it will depend on the length of the delimiter - unlike the initial character recognition, this is a complete state. Again, the value supplied for i should be the string length of the delimiter.

⟨*1stend*⟩≡
```
  (eval-when (:compile-toplevel :load-toplevel)
  (defun expand1stend (i)
    (if (> i 1)
        (list
         `,(intern (format nil "CHUNK-START-TAG-~A" (- i 1)) webpkg)
         '(incf pos)
         '(setf code (aref buff pos))
         `(if (eql code ,(intern (format nil "START-CHUNK-CHAR-~A" i) webpkg))
           (progn
             (setf start-pos (+ pos 1))
             (go in-chunk-name)))
         '(if (eql code newline-code)
           (go normal-start))
         '(go normal))))
  )
```

If the length of the string is 3 or greater, the states for the middle characters must also be generated. In this case the standard code for returning to normal scanning in the case of an incorrect character or newline is included, as the complete state machinery for the inter-character states must be autogenerated.

⟨*1stmiddle*⟩≡
```
  (eval-when (:compile-toplevel :load-toplevel)
  (defun expand1stmiddle (i)
    (if (> i 2)
        (loop for j from 2 to (- i 1) append
              (list
               (intern (format nil "CHUNK-START-TAG-~A" (- j 1)) webpkg)
               '(incf pos)
               '(setf code (aref buff pos))
               `(if (eql code ,(intern (format nil "START-CHUNK-CHAR-~A" j) webpkg ))
                 (go ,(intern (format nil "CHUNK-START-TAG-~A" j) webpkg)))
               '(if (eql code newline-code)
                 (go normal-start))
               '(go normal)))))
  )
```

8.2.2 2nd - the End of a Chunk Name and Beginning of Chunk Content

As with the start of the 1st delimiter, the initial recognition of the 2nd delimiter takes place in another state and does not need its own state name. In the case of a single character, the initial recognition is the only condition needed to satisfy the end of the chunk name. This means the complete position of the chunk name is known and the character sequence containing the name is assigned to a variable. At this point a design decision to allow characters in multi-character delimiter strings to also be present as chunk names is expressed by having multi-string states return to the in-chunk-name state rather than normal reading.

⟨*2ndstart*⟩≡
```
(eval-when (:compile-toplevel :load-toplevel)
(defun expand2ndstart (i)
  (if (= 1 i)
      `(if (eql code end-name-char-1)
        (progn
          (setf chunk-name (subseq buff start-pos (- pos (length chunk-ref-end-delimiter))))
          (go after-chunk-name)))
      `(if (eql code end-name-char-1)
        (go chunk-name-end-tag-1))))
)
```

If the length is greater than one, the assignment of the chunk name comes later:

⟨*2ndend*⟩≡
```
(eval-when (:compile-toplevel :load-toplevel)
(defun expand2ndend (i)
  (if (> i 1)
      (list
      `,(intern (format nil "CHUNK-NAME-END-TAG-~A" (- i 1)) webpkg)
      '(incf pos)
      '(setf code (aref buff pos))
      `(if (eql code ,(intern (format nil "END-NAME-CHAR-~A" i) webpkg))
        (progn
          (setf chunk-name (subseq buff start-pos
                                   (- pos (length chunk-ref-end-delimiter))))
          (go after-chunk-name)))
      '(if (eql code newline-code)
        (go normal-start))
      '(go in-chunk-name))))
)
```

For delimiter lengths greater than two intermediate states are needed:

⟨*2ndmiddle*⟩≡
```
(eval-when (:compile-toplevel :load-toplevel)
(defun expand2ndmiddle (i)
  (if (> i 2)
      (loop for j from 2 to (- i 1) append
            (list
             (intern (format nil "CHUNK-NAME-END-TAG-~A" (- j 1)) webpkg)
             '(incf pos)
             '(setf code (aref buff pos))
             `(if (eql code ,(intern (format nil "END-NAME-CHAR-~A" j) webpkg ))
               (go ,(intern (format nil "CHUNK-NAME-END-TAG-~A" j) webpkg)))
             '(if (eql code newline-code)
               (go normal-start))
             '(go in-chunk-name)))))
)
```

8.2.3 3rd - Start of a Chunk Reference

Inside of a chunk, there may be references to other chunks intended for insertion into the body of the current chunk. The necessary and sufficient task here is to identify the name of the chunk and the position of insertion - the rest is handled later.

⟨*3rdstart*⟩≡
```
(eval-when (:compile-toplevel :load-toplevel)
(defun expand3rdstart (i)
  (if (= 1 i)
      `(if (eql code start-ref-char-1)
        (progn
          (setf start-pos (+ pos 1))
          (go in-chunk-ref)))
      `(if (eql code start-ref-char-1)
        (go start-ref-tag-1))))
)
```

⟨*3rdend*⟩≡
```
(eval-when (:compile-toplevel :load-toplevel)
(defun expand3rdend (i)
  (if (> i 1)
      (list
       `,(intern (format nil "START-REF-TAG-~A" (- i 1)) webpkg)
       '(incf pos)
       '(setf code (aref buff pos))
       `(if (eql code ,(intern (format nil "START-REF-CHAR-~A" i) webpkg))
         (progn
           (setf start-pos (+ pos 1))
           (go in-chunk-ref)))
       '(if (eql code newline-code)
         (go chunk-start))
       '(go in-chunk))))
)
```

⟨*3rdmiddle*⟩≡
```
(eval-when (:compile-toplevel :load-toplevel)
(defun expand3rdmiddle (i)
  (if (> i 2)
      (loop for j from 2 to (- i 1) append
            (list
             (intern (format nil "START-REF-TAG-~A" (- j 1)) webpkg)
             '(incf pos)
             '(setf code (aref buff pos))
             `(if (eql code ,(intern (format nil "START-REF-CHAR-~A" j) webpkg ))
               (go ,(intern (format nil "START-REF-TAG-~A" j) webpkg)))
             '(if (eql code newline-code)
               (go chunk-start))
             '(go in-chunk)))))
)
```

8.2.4 4th - End of a Chunk Reference

The recognition of the end of a chunk is essentially the same as recognizing the beginning from a matching standpoint. Once all information is known, the position of the reference in the current chunk and the string identifying the chunk being referenced are written to the list holding the structural information for the current chunk being read. Here again, as in the end of chunk name case, characters that are part of multi-character delimiters are permitted in chunk names.

⟨*4thstart*⟩≡
```
(eval-when (:compile-toplevel :load-toplevel)
(defun expand4thstart (i)
  (if (= 1 i)
      `(if (eql code end-ref-char-1)
        (progn
          (push (list chunk-start-pos (- start-pos (length chunk-ref-end-delimiter))) content)
          (push (array-to-string (subseq buff start-pos (- pos 1)))
                content)
          (setf chunk-start-pos (+ pos 1))
          (go in-chunk)))
      `(if (eql code end-ref-char-1)
        (go end-ref-tag-1))))
)
```

⟨*4thend*⟩≡

```
  (eval-when (:compile-toplevel :load-toplevel)
  (defun expand4thend (i)
    (if (> i 1)
        (list
         `,(intern (format nil "END-REF-TAG-~A" (- i 1)) webpkg)
         '(incf pos)
         '(setf code (aref buff pos))
         `(if (eql code ,(intern (format nil "END-REF-CHAR-~A" i) webpkg))
           (progn
             (push (list chunk-start-pos (- start-pos (length chunk-ref-end-delimiter))) content)
             (push (array-to-string
                    (subseq buff start-pos (- pos 1))) content)
             (setf chunk-start-pos (+ pos 1))
             (go in-chunk)))
         '(if (eql code newline-code)
           (go chunk-start))
         '(go in-chunk-ref))))
  )
```

⟨*4thmiddle*⟩≡

```
  (eval-when (:compile-toplevel :load-toplevel)
  (defun expand4thmiddle (i)
    (if (> i 2)
        (loop for j from 2 to (- i 1) append
              (list
               (intern (format nil "END-REF-TAG-~A" (- j 1)) webpkg)
               '(incf pos)
               '(setf code (aref buff pos))
               `(if (eql code ,(intern (format nil "END-REF-CHAR-~A" j) webpkg ))
                 (go ,(intern (format nil "END-REF-TAG-~A" j) webpkg)))
               '(if (eql code newline-code)
                 (go chunk-start))
               '(go in-chunk-ref)))))
  )
```

8.2.5 5th - End of a Chunk

The end of the chunk is where all information is known. The last positonal information is written to the contents slot in the current chunk's structure (this will identify the entire chunk body if no references were contained in it, otherwise this will identify the region from the last chunk reference to the end of the chunk.

⟨*5thstart*⟩≡
```
(eval-when (:compile-toplevel :load-toplevel)
(defun expand5thstart (i)
  (if (= 1 i)
      `(if (eql code end-chunk-char-1)
        (progn
          (push (list chunk-start-pos chunk-end-pos) content)
          (setf content (reverse content))
          (add-to-chunk-contents (array-to-string chunk-name)
                                 content)
          (setf content nil)
          (go after-chunk-body)))
      `(if (eql code end-chunk-char-1)
        (go end-chunk-tag-1))))
)
```

⟨*5thend*⟩≡
```
(eval-when (:compile-toplevel :load-toplevel)
(defun expand5thend (i)
  (if (> i 1)
      (list
       `,(intern (format nil "END-CHUNK-TAG-~A" (- i 1)) webpkg)
       '(incf pos)
       '(setf code (aref buff pos))
       `(if (eql code ,(intern (format nil "END-CHUNK-CHAR-~A" i) webpkg))
         (progn
           (push (list chunk-start pos chunk-end-pos) content)
           (setf content (reverse content))
           (add-to-chunk-contents (array-to-string chunk-name)
                                  content)
           (setf content nil)
           (go after-chunk-body)))
       '(if (eql code newline-code)
         (go chunk-start))
       '(go in-chunk))))
)
```

⟨*5thmiddle*⟩≡
```
(eval-when (:compile-toplevel :load-toplevel)
(defun expand5thmiddle (i)
  (if (> i 2)
      (loop for j from 2 to (- i 1) append
            (list
             (intern (format nil "END-CHUNK-TAG-~A" (- j 1)) webpkg)
             '(incf pos)
             '(setf code (aref buff pos))
             `(if (eql code ,(intern (format nil "END-CHUNK-CHAR-~A" j) webpkg ))
               (go ,(intern (format nil "END-CHUNK-TAG-~A" j) webpkg)))
             '(if (eql code newline-code)
               (go chunk-start))
             '(go in-chunk)))))
)
```

8.3 Generating The Scan Function

When generating a source code document using `tangle`, it is useful to retain some knowledge of the positioning of elements in the original file. Therefore each chunk structure definition has a line position which records the original line in the pamphlet on which the line is found. This means that the scanning code must keep track of how many newlines it has seen. In addition, it must maintain its position information and store key positions for chunks it is in the process of scanning. This is accomplished with “global” variables within the context of the scan, which each state will update.

Broadly, the macro to generate the scanning function can be defined as follows:

⟨*scan-toplevel*⟩≡
```
(defmacro generate-scan-function ()
  `(defun scan-for-chunks (buff)
    (declare (optimize (speed 3)))
    ,(append '(prog)
       ⟨initialization⟩
       ⟨normal-start⟩
       ⟨normal⟩
       ⟨start-chunk⟩
       ⟨in-chunk-name⟩
       ⟨end-of-name⟩
       ⟨clear-spaces-after-name⟩
       ⟨in-chunk-body⟩
       ⟨new-chunk-line⟩
       ⟨end-of-chunk⟩
       ⟨after-chunk-body⟩
       ⟨start-chunk-ref⟩
       ⟨in-chunk-ref⟩
       ⟨end-chunk-ref⟩
      )))
```

It is important to be sure that the generated names below are defined in the correct package (`cl-web`, in this case) and to be sure of that assignment the variable webpkg is used to explicitly specify where to intern the generated symbols.

8.4 Initialization

Initialization contains several “setup” steps carried out prior to any scanning. They involve delcaring positional variables and other local variables, setting optimization options for the Lisp compiler, and defining the types of key local variables in order to allow the compiler to translate them more efficiently.

The purpose of the variables:

- `pos` - The current scan position in the file. Initialized to `-1` in order to be incremented to zero, the first actual array position, when the scan starts.
- `end-buff` - The array position corresponding to the last character in the file.
- `start-pos` - This holds the starting position for either the chunk name at the beginning of a chunk definition or the chunk name in a chunk reference (depending on the state of the scan.)
- `code` - The character currently being considered by the scan
- `chunk-name` - The name of the chunk currently being scanned.
- `content` - The list of character positions and chunk names defining the chunk currently being scanned.
- `chunk-start-pos` - The starting point of the current chunk body, a.k.a the first non-space non-newline character after a legal chunk name.
- `chunk-end-pos` - The last position before the end of chunk body tag, a.k.a the end position of the code inside the current chunk.

⟨*initialization*⟩≡
```
            '(((pos -1)
              (end-buff (- (array-dimension buff 0) 1))
              (start-pos)
              (code)
              (chunk-name)
              (content nil)
              (chunk-start-pos)
              (chunk-end-pos)
              (line-number 0)))
           '((declare (type (signed-byte 32) pos end-buff line-number)
                   (type (simple-array (unsigned-byte  8) (*)) buff)))
```

8.5 Normal Start

Normal start is both the starting point for all non-chunk new lines and the starting point for parsing the file itself. The first step is to increment the position (this is why the start postion is -1, to put this first read at position 0 - in lisp the starting position of an array is 0 not 1). Line number is also incremented, and then a check is made for the scan having run either beyond or to the end of the file. If it has, the scan is complete and a return call is issued. Otherwise, code is set to the character at the current position. If this code matches the first code in the start of chunk tag, `go` is used to jump to the code defined for that purpose (later in this document). If a newline code is found, normal-start returns to the beginning and starts over on the next line. All other characters are normal in this context and trigger the "normal" branch.

The strings chosen for tags make a difference in which action is taken even at this stage. If the string identifying the start of a chunk is only one character long (uncommon but possible) the finite state diagram earlier simplifies and there is no need to identify intermediate or ending tag characters.

⟨*normal-start*⟩≡
```
'(normal-start
(incf pos)
(incf line-number)
(if (>= pos end-buff)
    (return-from scan-for-chunks))
(setf code (aref buff pos)))
(list (expand1ststart (length chunk-start-delimiter)))
'((if (eql code newline-code)
    (go normal-start))
(go normal))
```

8.6 Normal Reading

`normal` is the state the scan process is in whenever it is not dealing with a chunk, noline character, or the end of the file. This function does not deal with an abrupt ending to a file (i.e. no newline character at the end of the file) because the original read function appended a newline character to the file as it was converted to an array. This guarantees the last character in the file will always be a newline character. The assumption made earlier that all chunk names are on the first column of a new line means that no checking for special tags is needed beyond the first character of a line. Therefore, in this state, that only the newline and non-newline character cases need to be delt with.

⟨*normal*⟩≡
```
'(normal
(incf pos)
(setf code (aref buff pos))
(if (eql code newline-code)
    (go normal-start))
(go normal))
```

8.7 Starting a Chunk and Handling Arbitrary Tag Lengths

If the `normal-start` condition identifies a character which matches the first character of the tag previously defined as the identifier for the start of a chunk, and the tag length is greater than one character, the next step is to identify if the second character in the tag is present in the next position. This subsection does not actually constitute the start of chunk parsing for the single character case, but in all other cases this is the first state to be reached.

If the tag is only two characters long, from this point the next state is the actual chunk name. Otherwise, the next character must be checked for, and so on for as many chars as constitute the tag. Rather than hand write all of the code to do this, a lisp macro is used to generate the necessary code.

Once the final character is spotted, the starting point of the name of the chunk is known and is assigned to the variable start-pos.

⟨*start-chunk*⟩≡
```
(expand1stmiddle (length chunk-start-delimiter))
(expand1stend (length chunk-start-delimiter))
```

8.8 Inside the Chunk Name

At this state our options are much the same, except in this case we are looking for the first char matching the end of name tag's first char. If we don't find it before the end of the line the line is treated as a normal line. If the ending tag is a single char in length finding it moves directly to checking for spaces, otherwise the check is for the next char in the tag.

⟨*in-chunk-name*⟩≡
```
'(in-chunk-name
 (incf pos)
 (setf code (aref buff pos)))
 (list (expand2ndstart (length chunk-name-end-delimiter)))
'((if (eql code newline-code)
     (go normal-start))
 (go in-chunk-name))
```

8.9 Exiting the Chunk Name and Handling Arbitrary Tag Lengths

If the delimiter length of the end-of-chunk-name tag is greater than 1, these functions will expand into code.

⟨*end-of-name*⟩≡
```
(expand2ndmiddle (length chunk-name-end-delimiter))
(expand2ndend (length chunk-name-end-delimiter))
```

8.10 Avoiding Spaces After Chunk Name

It is undesirable to include excess spaces which may be present after a chunk name, as the chunk contents are presumed to begin on the next line. This state skips over spaces if present. If any non-space non-newline characters are present, the chunk name is abandoned as invalid and the state returns to normal.

⟨*clear-spaces-after-name*⟩≡

```
'(after-chunk-name
(incf pos)
(setf code (aref buff pos))
(if (eql code newline-code)
    (progn
      (setf chunk-start-pos (+ pos 1))
      (go chunk-start)))
(if (eql code space-code)
    (go after-chunk-name))
(go normal))
```

8.11 Inside the Chunk Body

Inside a chunk body, the two important characters to watch for are the first character of a chunk reference, and a newline. No other significant characters are recognized in normal chunk scanning.

⟨*in-chunk-body*⟩≡

```
'(in-chunk
(incf pos)
(setf code (aref buff pos))
(if (eql code newline-code)
    (go chunk-start)))
(list (expand3rdstart (length chunk-ref-start-delimiter)))
'((go in-chunk))
```

8.12 New Chunk Line

New chunk lines require checks for several states. A check for an empty chunk is needed - if the start position is greater than the ending position an empty chunk is assumed and the positions are made equal. A newline means an immediate move to the next character and a return to the top of the chunk-start state. It is also necessary to check for reference tags and end of chunk tags in this state, but the details of doing so are dependent on the delimiter strings used and so function calls are made to the previously defined functions for 3rd and 5th delimiters.

⟨*new-chunk-line*⟩≡

```
'(chunk-start
(setf chunk-end-pos pos)
(if (> chunk-start-pos chunk-end-pos)
    (setf chunk-end-pos chunk-start-pos))
(incf pos)
(if (>= pos end-buff)
    (break "unexpected end of file"))
(setf code (aref buff pos))
(incf line-number)
(if (eql code newline-code)
    (go chunk-start)))
(list (expand3rdstart (length chunk-ref-start-delimiter)))
(list (expand5thstart (length chunk-end-delimiter)))
'((go in-chunk))
```

8.13 Exiting the End of Chunk Tag and Handling Arbitrary Tag Lengths

Again, details of delimiter handling need to be dynamically generated for situations where the end of chunk delimiter is longer than one character.

⟨*end-of-chunk*⟩≡

```
(expand5thmiddle (length chunk-end-delimiter))
(expand5thend (length chunk-end-delimiter))
```

8.14 Garbage after Chunk Name

As with the chunk name, it is assumed that there may be excess spaces after an end of chunk delimiter. These are ignored, but any non-space non-newline character will trigger an error exiting from the scan.

⟨*after-chunk-body*⟩≡

```
'(after-chunk-body
 (incf pos)
 (setf code (aref buff pos))
 (if (eql code newline-code)
     (go normal-start))
 (if (eql code space-code)
     (go after-chunk-body))
 (format t "pos: ~A, code: ~A~&" pos code)
 (break "garbage after end of chunk marker"))
```

8.15 Chunk Reference Starting Tags Longer than 1 Char

Standard function calls to generate states for chunk-start-reference delimiters longer than one character.

⟨*start-chunk-ref*⟩≡

```
(expand3rdmiddle (length chunk-ref-start-delimiter))
(expand3rdend (length chunk-ref-start-delimiter))
```

8.16 Inside a Chunk Reference

Inside a chunk reference, the only job is to look for the end of the reference or an unexpected newline (newlines are not legal in reference names.) Also present is a function call to generate the correct code to watch for the end of the reference.

⟨*in-chunk-ref*⟩≡

```
'(in-chunk-ref
 (incf pos)
 (setf code (aref buff pos)))
 (list (expand4thstart (length chunk-ref-end-delimiter)))
 '((if (eql code newline-code)
     (go chunk-start))
 (go in-chunk-ref))
```

8.17 Chunk Reference Ending Tags

Function calls to handle chunk-end-reference delimiters longer than 1 character.

⟨*end-chunk-ref*⟩≡

```
(expand4thmiddle (length chunk-ref-end-delimiter))
(expand4thend (length chunk-ref-end-delimiter))
```

9 Generating the Scan Function

Once all aspects of the scan operation have been defined and the constants which make up the delimiters are know, the scan-for-chunks function must be generated. This is handled in two ways. If the file is being compiled, we want to evaluate this function to generate the scan-for-chunks for the compiled file. We also want it to run if we are loading the file uncompiled, but we do NOT want it to run if we are loading the compiled file because (thanks to the eval-when) we have already done so in that case. So the first eval-when is to make sure the scan function is compiled with the file, and the second is to build it if it is not present.

⟨*generate-scan-function*⟩≡

```
  (eval-when (:compile-toplevel) (generate-scan-function))
  (if (not (fboundp 'scan-for-chunks)) (generate-scan-function))
```

10 Output to File

Once full information is available on chunk structure, it becomes possible to write out just the source code. However, because the scan did not in itself create any new sub-files (either in memory or on a storage medium) a command that can use the information in the hash table to generate the source file is needed. In theory, any chunk may be selected as the "top-level" chunk for a destination file, and so a command `write-chunk` is defined:

⟨*write-chunk*⟩≡

```
  (defun write-chunk (buff name out-file)
      (let ((chunk (gethash name *chunk-hash-table*)))
          (dolist (segs (reverse (chunk-contents chunk)))
              (dolist (seg segs)
                  (if (consp seg)
                      (write-sequence buff out-file :start (nth 0 seg)
                                                    :end (nth 1 seg))
                      (write-chunk buff seg out-file))))))
```

11 The Top-Level Command: `tangle`

Now that the necessary capabilities have been defined, it remains only to call them in the correct order. Briefly:

1. Clear the hash table
2. Read in the input file to an array
3. Scan the input array and record chunk information
4. Write the contents of the requested chunk and referenced chunks to the specified output file.

This version of `tangle` will always overwrite pre-existing files. While NOWEB will output all chunks in a file, using as a default output name the chunk name, this `tangle` command will output only the specified chunks and abandon the rest. Possible desirable behavior would be to warn if there are chunks un-extracted, or allow multiple chunks to multiple files with a single scan. If no chunk name is specified, the `*` character is assumed to denote the desired chunk to extract.

⟨*tangle*⟩≡
```
(defun tangle (in-file out-name &optional (chunkname "*"))
    (clrhash *chunk-hash-table*)
    (let ((buff (read-pamphlet-file in-file))
          (chunk)
          (name (make-array (list 1)
                                        :element-type '(unsigned-byte  8)
                                        :initial-element (char-code #\*))))
          (setf name chunkname)
          (scan-for-chunks buff)
          (setf chunk (gethash name *chunk-hash-table*))
          (if (not chunk)
              (break "No main chunk"))
          (with-open-file (out-file out-name :direction :output
                                       :if-exists :supersede
                                       :element-type '(unsigned-byte  8))
              (write-chunk buff name out-file))))
```

12 Debugging and Utility Functions

These functions are not normally needed for the `tangle` operation, but are useful when trying to diagnose problems.[7]

⟨*viewall*⟩≡
```
(defun view-all-chunks ()
  (maphash #'(lambda (k v) (format t "~a => ~a~%" k v)) *chunk-hash-table*))
```

⟨*viewchunk*⟩≡
```
(defun view-chunk (chunkname)
  (format t "~a => ~a~%" chunkname (gethash chunkname *chunk-hash-table*)))
```

[7]Should really add some more "friendly" view-chunk options here.

13 The Top Level File Structure

Now that all pieces are defined, the last step is to order them in the correct order for machine evaluation.

```
⟨*⟩≡
  ⟨copyright⟩
  ⟨license⟩
  ⟨package-def⟩
  ⟨array-to-string⟩
  ⟨hashtable⟩
  ⟨first-constants⟩
  ⟨tangle-delimiter-variables⟩
  ⟨generate-marker-constants⟩
  ⟨make-axiom-constants⟩
  ⟨chunk-structure⟩
  ⟨add-to-chunk-contents⟩
  ⟨file-to-array⟩
  ⟨read-pamphlet-file⟩
  ⟨1ststart⟩
  ⟨1stend⟩
  ⟨1stmiddle⟩
  ⟨2ndstart⟩
  ⟨2ndend⟩
  ⟨2ndmiddle⟩
  ⟨3rdstart⟩
  ⟨3rdend⟩
  ⟨3rdmiddle⟩
  ⟨4thstart⟩
  ⟨4thend⟩
  ⟨4thmiddle⟩
  ⟨5thstart⟩
  ⟨5thend⟩
  ⟨5thmiddle⟩
  ⟨scan-toplevel⟩
  ⟨generate-scan-function⟩
  ⟨write-chunk⟩
  ⟨tangle⟩
  ⟨viewall⟩
  ⟨viewchunk⟩
```