1. Introduction. This is `CLWEB`, a literate programming system for Common Lisp by Alex Plotnick ⟨*plotnick@cs.brandeis.edu*⟩. It is modeled after the `CWEB` system by Silvio Levy and Donald E. Knuth, which was in turn adapted from Knuth's original `WEB` system. It shares with those systems not only their underlying philosophy, but also most of their syntax. Readers unfamiliar with either of them—or with literate programming in general—should consult the `CWEB` manual or Knuth's «*Literate Programming*» (CSLI: 1992).

This is a priliminary, β-quality release of the system. To obtain the latest version, please visit `http://www.cs.brandeis.edu/~plotnick/clweb/`.

2. A `CLWEB` source file consists of a mixture of TeX, Lisp, and `WEB` control codes, but which is primary depends on your point of view. The `CWEB` manual, for instance, says that "[w]riting `CWEB` programs is something like writing TeX documents, but with an additional 'C mode' that is added to TeX's horizontal mode, vertical mode, and math mode." The same applies, *mutatis mutandis,* to the current system, but one might just as easily think of a web as some code with documentation blocks and special control codes sprinkled throughout, or as a completely separate language containing blocks that happen to have the syntax (more or less) of TeX and Lisp. For the purposes of understanding the implementation, this last view is perhaps the most useful, since the control codes determine which syntax to use in reading the material that follows.

The syntax of the `CLWEB` control codes themselves is similar to that of dispatching reader macro characters in Lisp: they all begin with '`@`x', where x is a single character that selects the control code. Most of the `CLWEB` control codes are quite similar to the ones used in `CWEB`; see the `CWEB` manual for detailed descriptions of the individual codes.

3. A literate programming system provides two primary operations: *tangling* and *weaving*. The tangler prepares a literate program, or *web*, for evaluation by a machine, while the weaver prepares it for typesetting and subsequent reading by a human. These operations reflect the two uses of a literate program, and the two audiences by whom it must be read: the computer on the one hand, and the human programmers that must understand and maintain it on the other.

Our tangler has two main interface functions: *tangle-file* and *load-web*. The first is analogous to *compile-file*: given a file containing `CLWEB` source, it produces an output file that can be subsequently loaded into a Lisp image with *load*. The function *load-web* is analogous to *load*, but also accepts `CLWEB` source as input instead of ordinary Lisp source: it loads a web into the Lisp environment.

The weaver has a single entry point: *weave* takes a web as input and generates a file that can be fed to TeX to generate a pretty-printed version of that web.

4. We'll start by setting up a package for the system. In addition to the top-level tangler and weaver functions mentioned above, there's also *load-sections-from-temp-file*, which is conceptually part of the tangler, but is a special-purpose routine designed to be used in conjunction with an editor such as Emacs to provide incremental redefinition of sections; the user will generally never need to call it directly. The remainder of the exported symbols are condition classes for the various errors and warnings that might be signaled while processing a web.

```
(provide "CLWEB")
(eval-when (:compile-toplevel :load-toplevel :execute)
   #+:sbcl (require 'sb-cltl2))
(defpackage "CLWEB"
   (:use "COMMON-LISP"
         #+:sbcl "SB-CLTL2"
         #+:allegro "SYS")
   (:export "TANGLE-FILE"
            "LOAD-WEB"
            "WEAVE"
            "LOAD-SECTIONS-FROM-TEMP-FILE"
            "AMBIGUOUS-PREFIX-ERROR"
            "SECTION-NAME-CONTEXT-ERROR"
            "SECTION-NAME-USE-ERROR"
            "SECTION-NAME-DEFINITION-ERROR"
            "UNUSED-NAMED-SECTION-WARNING")
   (:shadow "ENCLOSE"
             #+:allegro "FUNCTION-INFORMATION"
             #+:allegro "VARIABLE-INFORMATION"))
(in-package "CLWEB")
```

5. We'll define our global variables and condition classes as we need them, but we'd like them to appear near the top of the tangled output.

⟨ Global variables 12 ⟩
⟨ Condition classes 27 ⟩

6. Here's a little utility function that we'll use often. It's particularly useful for functions that accept a list desginator.

```
(defun ensure-list (object)
   (if (listp object) object (list object)))
```

7. And here's one taken from PCL: *mapappend* is like *mapcar* except that the results are appended together.

```
(defun mapappend (function &rest args)
   (if (some #'null args)
      ()
      (append (apply function (mapcar #'car args))
              (apply #'mapappend function (mapcar #'cdr args)))))
```

8. Sections. The fundamental unit of a web is the *section*, which may be either *named* or *unnamed*. Named sections are conceptually very much like parameterless macros, except that they can be defined piecemeal. The tangler replaces references to a named section with all of the code defined in all of the sections with that name. (This is where the name 'tangling' comes from: code may be broken up and presented in whatever order suits the expository purposes of the author, but is then spliced back together into the order that the programming language expects.) Unnamed sections, on the other hand, are evaluated or written out to a file for compilation in the order in which they appear in the source file.

Every section is assigned a number, which the weaver uses for generating cross-references. The numbers themselves never appear in the source file: they are generated automatically by the system.

Aside from a name, a section may have a *commentary part*, optionally followed by a *code part.* (We don't support the 'middle' part of a section that `WEB` and `CWEB`'s sections have, since the kinds of definitions that can appear there are essentially irrelevant in Lisp.) The commentary part consists of TEX material that describes the section; the weaver copies it (nearly) verbatim into the TEX output file, and the tangler ignores it. The code part contains Lisp forms and named section references; the tangler will eventually evaluate or compile those forms, while the weaver pretty-prints them to the TEX output file.

Three control codes begin a section: `@␣`, `@*`, and `@t`. Most sections will begin with `@␣`: these are 'regular' sections, which might be named or unnamed.

```
(defclass section ()
  ((name :accessor section-name :initarg :name)
   (number :accessor section-number)
   (commentary :accessor section-commentary :initarg :commentary)
   (code :accessor section-code :initarg :code))
  (:default-initargs :name nil :commentary nil :code nil))
```

9. Sections introduced with `@*` ('starred' sections) begin a new major group of sections, and get some special formatting during weaving. The control code `@*` should be immediately followed by a title for this group, terminated by a period. That title will appear as a run-in heading at the beginning of the section, as a running head on all subsequent pages until the next starred section, and in the table of contents.

The tangler does not care about the distinction between sections with stars and ones with none upon thars.

```
(defclass starred-section (section) ())
```

10. Sections that begin with `@t` are *test sections.* They are used to include test cases alongside the normal code, but are treated specially by both the tangler and the weaver. The tangler writes them out to a separate file, and the weaver may elide them entirely.

Test sections are automatically associated with the last non-test section defined, on the assumption that tests will be defined immediately after the code they're designed to exercise.

```
(defclass test-section (section)
  ((test-for :accessor test-for-section :initform *current-section*)))
(defclass starred-test-section (test-section starred-section) ())
```

11. There can also be TEX text preceding the start of the first section (i.e., before the first `@␣` or `@*`), called *limbo text.* Limbo text is generally used to define document-specific formatting macros, set up fonts, *&c.* The weaver passes it through virtually verbatim to the output file (only replacing occurrences of '`@@`' with '`@`'), and the tangler ignores it completely.

A single instance of the class *limbo-section* contains the limbo text in its *commentary* slot; it will never have a code part.

```
(defclass limbo-section (section) ())
```

12. Whenever we create a non-test section, we store it in the global **sections** vector and set its number to its index therein. This means that section objects won't be collected by the garbage collector even after the tangling or weaving has completed, but there's a good reason: keeping them around allows incremental redefinition of a web, which is important for interactive development.

We'll also keep the global variable **current-section** pointing to the last section (test or not) created.

⟨ Global variables 12 ⟩ ≡
```
(defvar *sections* (make-array 128 :adjustable t :fill-pointer 0))
(defvar *current-section* nil)
```
See also sections 15, 29, 32, 35, 39, 46, 49, 57, 59, 73, 127, 128, 210, 225, and 235.

This code is used in section 5.

13. ⟨ Initialize global variables 13 ⟩ ≡
```
(setf (fill-pointer *sections*) 0)
(setf *current-section* nil)
```
See also sections 16, 30, 211, and 226.

This code is used in sections 118, 121, and 125.

14. Here's where section numbers are assigned. We use a generic function for *push-section* so that we can override it for test sections.
```
(defgeneric push-section (section))
(defmethod push-section ((section section))
   (setf (section-number section) (vector-push-extend section *sections*))
   section)
(defmethod initialize-instance :after ((section section) &rest initargs &key)
   (declare (ignore initargs))
   (setq *current-section* (push-section section)))
```

15. Test sections aren't stored in the **sections** vector; we keep them separate so that they won't interfere with the numbering of the other sections.

⟨ Global variables 12 ⟩ +≡
```
(defvar *test-sections* (make-array 128 :adjustable t :fill-pointer 0))
```

16. ⟨ Initialize global variables 13 ⟩ +≡
```
(setf (fill-pointer *test-sections*) 0)
```

17.
```
(defmethod push-section ((section test-section))
   (let ((*sections* *test-sections*))
      (call-next-method)))
```

18. The test sections all get woven to a separate output file, and we'll need a copy of the limbo text there, too.
```
(defmethod push-section :after ((section limbo-section))
   (vector-push-extend section *test-sections*))
```

19. We keep named sections in a binary search tree whose keys are section names and whose values are code forms; the tangler will replace references to those names with the associated code. We use a tree instead of, say, a hash table so that we can support abbreviations (see below).

```
(defclass binary-search-tree ()
  ((key :accessor node-key :initarg :key)
   (left-child :accessor left-child :initarg :left)
   (right-child :accessor right-child :initarg :right))
  (:default-initargs :left nil :right nil))
```

20. The primary interface to the BST is the following routine, which attempts to locate the node with key *item* in the tree rooted at *root*. If it is not already present and the *:insert-if-not-found* argument is true, a new node is created with that key and added to the tree. The arguments *:predicate* and *:test* should be designators for functions of two arguments, both of which will be node keys; *:predicate* should return true iff its first argument precedes its second in the total ordering used for the tree, and *:test* should return true iff the two keys are to be considered equivalent.

Two values are returned: the node with key *item* (or *nil* if no such node was found and *:insert-if-not-found* is false), and a boolean representing whether or not the node was already in the tree.

```
(defgeneric find-or-insert (item root &key predicate test insert-if-not-found))
(defmethod find-or-insert (item (root binary-search-tree) &key
                           (predicate #'<) (test #'eql)
                           (insert-if-not-found t))
  (flet ((lessp (item node) (funcall predicate item (node-key node)))
         (samep (item node) (funcall test item (node-key node))))
    (do ((parent nil node)
         (node root (if (lessp item node)
                        (left-child node)
                        (right-child node))))
        ((or (null node) (samep item node))
         (if node
             (values node t)
             (if insert-if-not-found
                 ⟨Insert a new node with key item and return it 21⟩
                 (values nil nil)))))))
```

21. ⟨Insert a new node with key *item* and return it 21⟩ ≡

```
(let ((node (make-instance (class-of root) :key item)))
  (when parent
    (if (lessp item parent)
        (setf (left-child parent) node)
        (setf (right-child parent) node)))
  (values node nil))
```

This code is used in section 20.

22. Besides searching, probably the most common operation on a BST is to traverse it in-order, applying some function to each node.

```
(defgeneric maptree (function tree))

(defmethod maptree (function (tree (eql nil)))
  (declare (ignore function)))
(defmethod maptree (function (tree binary-search-tree))
  (maptree function (left-child tree))
  (funcall function tree)
  (maptree function (right-child tree)))
```

23. As mentioned above, named sections can be defined piecemeal, with the code spread out over several sections in the `CLWEB` source. We might think of a named section as a sort of 'virtual' section, which consists of a name, the combined code parts of all of the physical sections with that name, and the number of the first such section.

And that's what we store in the BST: nodes that look like sections, inasmuch as they have specialized *section-name*, *section-code*, and *section-number* methods, but are not actually instances of the class *section*. The commentary and code is stored in the *section* instances that comprise a given named section: references to those sections are stored in the *sections* slot.

The weaver uses the last two slots, *used-by* and *cited-by*, to generate cross-references. They will be populated during reading with lists of all the sections that reference this named section.

```
(defclass named-section (binary-search-tree)
  ((key :accessor section-name :initarg :name)
   (sections :accessor named-section-sections :initform '())
   (used-by :accessor used-by :initform '())
   (cited-by :accessor cited-by :initform '())))

(defmethod named-section-sections :around ((section named-section))
  (sort (copy-list (call-next-method)) #'< :key #'section-number))

(defmethod section-code ((section named-section))
  (mapappend #'section-code (named-section-sections section)))

(defmethod section-number ((section named-section))
  (section-number (first (named-section-sections section))))
```

24. Section names in the input file can be abbreviated by giving a prefix of the full name followed by '...': e.g., `@<Frob...@>` might refer to the section named 'Frob *foo* and tweak *bar*'.

Here's a little utility routine that makes working with such section names easier. Given a name, it returns two values: true or false depending on whether the name is a prefix or not, and the length of the non-'`...`' segment of the name.

```
(defun section-name-prefix-p (name)
  (let ((len (length name)))
    (if (string= name "..." :start1 (max (- len 3) 0) :end1 len)
        (values t (- len 3))
        (values nil len))))
```

25. Next we need some special comparison routines for section names that might be abbreviations. We'll use these as the *:test* and *:predicate* functions, respectively, for our BST.

```
(defun section-name-lessp (name1 name2)
   (let ((len1 (nth-value 1 (section-name-prefix-p name1)))
         (len2 (nth-value 1 (section-name-prefix-p name2))))
      (string-lessp name1 name2 :end1 len1 :end2 len2)))
```

```
(defun section-name-equal (name1 name2)
   (multiple-value-bind (prefix-1-p len1) (section-name-prefix-p name1)
      (multiple-value-bind (prefix-2-p len2) (section-name-prefix-p name2)
         (let ((end (min len1 len2)))
            (if (or prefix-1-p prefix-2-p)
               (string-equal name1 name2 :end1 end :end2 end)
               (string-equal name1 name2))))))
```

26. When we look up a named section, either the name used to perform the lookup, the name for the section in the tree, or both might be a prefix of the full section name.

```
(defmethod find-or-insert (item (root named-section) &key
                                (predicate #'section-name-lessp)
                                (test #'section-name-equal)
                                (insert-if-not-found t))
   (multiple-value-bind (node present-p)
         (call-next-method item root
                           :predicate predicate
                           :test test
                           :insert-if-not-found insert-if-not-found)
      (if present-p
         (or ⟨ Check for an ambiguous match, and raise an error in that case 28 ⟩
             (values node t))
         (values node nil))))
```

27. ⟨ Condition classes 27 ⟩ ≡

```
(define-condition ambiguous-prefix-error (error)
   ((prefix :reader ambiguous-prefix :initarg :prefix)
    (first-match :reader ambiguous-prefix-first-match :initarg :first-match)
    (alt-match :reader ambiguous-prefix-alt-match :initarg :alt-match))
   (:report
    (λ (condition stream)
       (format stream "~@<Ambiguous␣prefix:␣<~A>␣matches␣both␣<~A>␣and␣<~A>~:@>"
               (ambiguous-prefix condition)
               (ambiguous-prefix-first-match condition)
               (ambiguous-prefix-alt-match condition)))))
```

See also sections 38, 99, 112, 123, and 179.

This code is used in section 5.

28. If there is an ambiguity in a prefix match, the tree ordering guarantees that it will occur in the sub-tree rooted at *node*.

⟨ Check for an ambiguous match, and raise an error in that case 28 ⟩ ≡

```
(dolist (child (list (left-child node) (right-child node)))
   (when child
      (multiple-value-bind (alt present-p)
            (call-next-method item child
                              :predicate predicate
                              :test test
                              :insert-if-not-found nil)
         (when present-p
            (restart-case
                  (error 'ambiguous-prefix-error
                        :prefix item
                        :first-match (node-key node)
                        :alt-match (node-key alt))
               (use-first-match ()
                  :report "Use␣the␣first␣match."
                  (return (values node t)))
               (use-alt-match ()
                  :report "Use␣alternate␣match."
                  (return (values alt t)))))))))
```

This code is used in section 26.

29. We store our named section tree in the global variable ∗*named-sections*∗, which is reset before each tangling or weaving. The reason this is global is the same as the reason ∗*sections*∗ was: to allow incremental redefinition.

⟨ Global variables 12 ⟩ +≡

```
(defvar ∗named-sections∗ nil)
```

30. ⟨ Initialize global variables 13 ⟩ +≡

```
(setq ∗named-sections∗ nil)
```

31. Section names are normalized by *squeeze*, which trims leading and trailing whitespace and replaces all runs of one or more whitespace characters with a single space.

```
(defun whitespacep (char) (find char ∗whitespace∗ :test #'char=))

(defun squeeze (string)
   (loop with squeezing = nil
         for char across (string-trim ∗whitespace∗ string)
         if (not squeezing)
           if (whitespacep char)
             do (setq squeezing t) and collect #\␣ into chars
           else
             collect char into chars
         else
           unless (whitespacep char)
             do (setq squeezing nil) and collect char into chars
         finally (return (coerce chars 'string))))
```

32. This list should contain the characters named ‘Space’, ‘Tab’, ‘Newline’, ‘Linefeed’, ‘Page’, and ‘Return’, in that order. However, ‘Linefeed’ might be the same character as ‘Newline’ or ‘Return’, and so might not appear as a distinct character. This is a known bug, caused by the fact that we’re not currently overriding the character name reader.

```
⟨ Global variables 12 ⟩ +≡
(defparameter *whitespace*
  #.(coerce '(#\␣ #\Tab #\Newline #\Newline #\Page #\Return) 'string))
```

33. The next routine is our primary interface to named sections: it looks up a section by name in the tree, and creates a new one if no such section exists.

```
(defun find-section (name &aux (name (squeeze name)))
  (if (null *named-sections*)
      (values (setq *named-sections* (make-instance 'named-section :name name)) nil)
      (multiple-value-bind (section present-p)
            (find-or-insert name *named-sections*)
        (when present-p
          ⟨ Update the section name if the new one is better 34 ⟩)
        (values section present-p))))
```

34. We only actually update the name of a section in two cases: if the new name is not an abbreviation but the old one was, or if they are both abbreviations but the new one is shorter. (We only need to compare against the shortest available prefix, since we detect ambiguous matches.)

```
⟨ Update the section name if the new one is better 34 ⟩ ≡
(multiple-value-bind (new-prefix-p new-len)
      (section-name-prefix-p name)
  (multiple-value-bind (old-prefix-p old-len)
        (section-name-prefix-p (section-name section))
    (when (or (and old-prefix-p (not new-prefix-p))
              (and old-prefix-p new-prefix-p (< new-len old-len)))
      (setf (section-name section) name))))
```

This code is used in section 33.

35. Reading. We distinguish five distinct modes for reading. Limbo mode is used for TEX text that proceeds the first section in a file. TEX mode is used for reading the commentary that begins a section. Lisp mode is used for reading the code part of a section; inner-Lisp mode is for reading Lisp forms that are embedded within TEX material. And finally, restricted mode is used for reading material in section names and a few other places.

We use separate readtables for each mode, which are stored in ∗*readtables*∗ and accessed via *readtable-for-mode*. We add an extra readtable with key *nil* that stores a virgin copy of the standard readtable.

```
⟨ Global variables 12 ⟩ +≡
(eval-when (:compile-toplevel :load-toplevel :execute)
   (defparameter ∗modes∗ '(:limbo :tex :lisp :inner-lisp :restricted)))
(deftype mode () `(member ,@∗modes∗))

(defvar ∗readtables∗
   (loop for mode in (cons nil ∗modes∗)
         collect (cons mode (copy-readtable nil))))
```

36.

```
(defun readtable-for-mode (mode)
   (declare (type (or mode null) mode))
   (cdr (assoc mode ∗readtables∗)))
```

37. The following macro is just a bit of syntactic sugar for executing the given forms with ∗*readtable*∗ bound appropriately for the given mode.

```
(defmacro with-mode (mode &body body)
   `(let ((∗readtable∗ (readtable-for-mode ,mode)))
      ,@body))
```

38. Sometimes we'll have to detect and report errors during reading. This condition class and associated signaling function allow *format*-style error reporting.

```
⟨ Condition classes 27 ⟩ +≡
(define-condition simple-reader-error (reader-error simple-condition) ()
   (:report (λ (condition stream)
               (format stream "~S␣on␣~S:~%~?"
                       condition (stream-error-stream condition)
                       (simple-condition-format-control condition)
                       (simple-condition-format-arguments condition)))))

(defun simple-reader-error (stream control &rest args)
   (error 'simple-reader-error
          :stream stream
          :format-control control
          :format-arguments args))
```

39. We frequently need an object to use as the *eof-value* argument to *read*. It need not be a symbol; it need not even be an atom.

```
⟨ Global variables 12 ⟩ +≡
(defvar ∗eof∗ (make-symbol "EOF"))
```

40.

```
(defun eof-p (x) (eq x ∗eof∗))
(deftype eof () '(satisfies eof-p))
```

41. We'll occasionally need to know if a given character terminates a token or not. This function answers that question, but only approximately—if the user has frobbed the current readtable and set non-standard characters to whitespace syntax, *this routine will not yield the correct result.* There's unfortunately nothing that we can do about it portably, since there's no standard way of determining the syntax of a character or of obtaining a list of all the characters with a given syntax.

```
(defun token-delimiter-p (char)
  (declare (type character char))
  (or (whitespacep char)
      (multiple-value-bind (function non-terminating-p) (get-macro-character char)
        (and function (not non-terminating-p)))))
```

42. In some of the reading routines we'll define below, we need to be careful about reader macro functions that don't return any values. For example, if the input file contains '(#|...|#)', and we naïvely call *read* in the reader macro function for '#\(', an error will be signaled, since *read* will skip over the comment and invoke the reader macro function for '#\)'.

The solution is to peek at the next character in the input stream and manually invoke the associated reader macro function (if any), returning a list of the values returned by that function. If the next character is not a macro character, we just call *read* or *read-preserving-whitespace*, returning a list containing the single object so read.

```
(defun read-maybe-nothing-internal (read stream eof-error-p eof-value recursive-p)
  (multiple-value-list
   (let* ((next-char (peek-char nil stream nil nil recursive-p))
          (macro-fun (and next-char (get-macro-character next-char))))
     (cond (macro-fun
            (read-char stream)
            (call-reader-macro-function macro-fun stream next-char))
           (t (funcall read stream eof-error-p eof-value recursive-p))))))

(defun read-maybe-nothing (stream &optional (eof-error-p t) eof-value recursive-p)
  (read-maybe-nothing-internal #'read stream eof-error-p eof-value recursive-p))

(defun read-maybe-nothing-preserving-whitespace (stream &optional (eof-error-p t) eof-value recursive-p)
  (read-maybe-nothing-internal #'read-preserving-whitespace stream eof-error-p eof-value recursive-p))
```

43. In SBCL, most of the standard reader macro functions assume that they're being called in an environment where the global read buffer is bound and initialized. It would be nice if that wasn't the case and we could elide the following nonsense.

```
(defmacro with-read-buffer ((&rest args) &body body)
  (declare (ignorable args))
  #+:sbcl '(sb-impl::with-read-buffer ,args ,@body)
  #-:sbcl '(progn ,@body))

(defun call-reader-macro-function (fn stream char)
  (with-read-buffer ()
    (funcall fn stream char)))
```

44. We want the weaver to output properly indented code, but it's basically impossible to automatically indent Common Lisp code without a complete static analysis. And so we don't try. What we do instead is assume that the input is indented correctly, and try to approximate that on output; we call this process *indentation tracking*.

The way we do this is to record the the column number, or *character position*, of every Lisp form in the input, and use those positions to reconstruct the original indentation.

We'll define a *charpos stream* as an object that tracks the character position of an underlying stream. Note that these aren't instances of *stream* (and can't be, without relying on an extension to Common Lisp like Gray streams). But they contain a standard composite stream we'll call a *proxy stream* which is hooked up to the underlying stream whose position they're tracking, and it's these proxy streams that we'll pass around, so that the standard stream functions will all work.

```
(defclass charpos-stream ()
   ((charpos :initarg :charpos :initform 0)
    (proxy-stream :accessor charpos-proxy-stream :initarg :proxy)))
```

45. The GF *charpos* returns the current character position of a charpos stream. It relies on the last calculated character position (stored in the *charpos* slot) and a buffer that stores the characters input or output since the last call to *charpos*, retrieved with *get-charpos-stream-buffer*. Tabs in the underlying stream are interpreted as advancing the column number to the next multiple of ∗*tab-width*∗.

```
(defgeneric get-charpos-stream-buffer (stream))

(defgeneric charpos (stream))
(defmethod charpos ((stream charpos-stream))
   (with-slots (charpos) stream
      (loop for char across (get-charpos-stream-buffer stream)
            do (case char
                   (#\Tab (incf charpos (- *tab-width* (rem charpos *tab-width*))))
                   (#\Newline (setf charpos 0))
                   (t (incf charpos)))
            finally (return charpos))))
```

46. ⟨ Global variables 12 ⟩ +≡

```
(defvar *tab-width* 8)
```

47. For tracking the character position of an input stream, our proxy stream will be an echo stream that takes input from the underlying stream and sends its output to a string stream, which we'll use as our buffer.

```
(defclass charpos-input-stream (charpos-stream) ())

(defmethod shared-initialize :around ((instance charpos-input-stream) slot-names &rest initargs &key stream)
   (apply #'call-next-method instance slot-names
          (list* :proxy (make-echo-stream
                           stream
                           (make-string-output-stream
                            :element-type (stream-element-type stream)))
                 initargs)))

(defmethod get-charpos-stream-buffer ((stream charpos-input-stream))
   (get-output-stream-string
    (echo-stream-output-stream (charpos-proxy-stream stream))))
```

48. For the output stream case, our proxy stream is a broadcast stream to the given stream and a fresh string stream, again used as a buffer.

```
(defclass charpos-output-stream (charpos-stream) ())
(defmethod shared-initialize :around ((instance charpos-output-stream) slot-names &rest initargs &key stream)
   (apply #'call-next-method instance slot-names
          (list* :proxy (make-broadcast-stream
                         (make-string-output-stream
                          :element-type (stream-element-type stream))
                         stream)
                 initargs)))
(defmethod get-charpos-stream-buffer ((stream charpos-output-stream))
   (get-output-stream-string
    (first (broadcast-stream-streams (charpos-proxy-stream stream)))))
```

49. Because we'll be passing around the proxy streams, we need to manually maintain a mapping between them and their associated instances of *charpos-stream*.

⟨ Global variables 12 ⟩ +≡
```
(defvar *charpos-streams* (make-hash-table :test #'eq))
```

50.
```
(defmethod initialize-instance :after ((instance charpos-stream) &rest initargs &key)
   (declare (ignore initargs))
   (setf (gethash (charpos-proxy-stream instance) *charpos-streams*) instance))
```

51. The top-level interface to the charpos streams are the following two functions: *stream-charpos* retrieves the character position of the stream for which *stream* is a proxy, and *release-charpos-stream* deletes the reference to the stream maintained by the associated *charpos-stream* instance.

```
(defun stream-charpos (stream)
   (charpos (or (gethash stream *charpos-streams*)
                (error "Not␣tracking␣charpos␣for␣~S" stream))))
(defun release-charpos-stream (stream)
   (multiple-value-bind (charpos-stream present-p)
         (gethash stream *charpos-streams*)
      (cond (present-p
              (setf (charpos-proxy-stream charpos-stream) nil) ; release stream
              (remhash stream *charpos-streams*))
             (t (warn "Not␣tracking␣charpos␣for␣~S" stream)))))
```

52. Here are a few convenience methods for creating charpos streams. The *input-stream* and *output-stream* arguments are stream designators.

```
(defun make-charpos-input-stream (input-stream &key (charpos 0))
  (make-instance 'charpos-input-stream
                 :stream (case input-stream
                           ((t) *terminal-io*)
                           ((nil) *standard-input*)
                           (otherwise input-stream))
                 :charpos charpos))

(defun make-charpos-output-stream (output-stream &key (charpos 0))
  (make-instance 'charpos-output-stream
                 :stream (case output-stream
                           ((t) *terminal-io*)
                           ((nil) *standard-output*)
                           (otherwise output-stream))
                 :charpos charpos))
```

53. And finally, here are a couple of macros that make using them easy and trouble-free. They execute *body* in a lexical environment in which *var* is bound to a proxy stream the tracks the character position for *stream*.

```
(defmacro with-charpos-input-stream ((var stream &key (charpos 0)) &body body)
  `(let ((,var (charpos-proxy-stream (make-charpos-input-stream ,stream :charpos ,charpos))))
     (unwind-protect (progn ,@body)
       (release-charpos-stream ,var))))

(defmacro with-charpos-output-stream ((var stream &key (charpos 0)) &body body)
  `(let ((,var (charpos-proxy-stream (make-charpos-output-stream ,stream :charpos ,charpos))))
     (unwind-protect (progn ,@body)
       (release-charpos-stream ,var))))
```

54. Sometimes we'll want to look more than one character ahead in a stream. This macro lets us do so, after a fashion: it executes *body* in a lexical environment where *var* is bound to a stream whose input comes from *stream* and *rewind* is a local function that 'rewinds' that stream to its state prior to any reads executed in the body.

```
(defmacro with-rewind-stream ((var stream &optional (rewind 'rewind))
                              &body body &aux (out (gensym)))
  `(with-open-stream (,out (make-string-output-stream))
     (with-open-stream (,var (make-echo-stream ,stream ,out))
       (flet ((,rewind ()
                (setq ,var (make-concatenated-stream
                            (make-string-input-stream (get-output-stream-string ,out))
                            ,var))))
         ,@body))))
```

55. And sometimes, we'll want to call *read* on a stream, and keep a copy of the characters that *read* actually scans. This macro reads from *stream*, then executes the *body* forms with *object* bound to the object returned by *read* and *echoed* bound to a variable containing the characters so consumed. If *prefix* is supplied, it should be a string that will be concatenated onto the front of *stream* prior to reading.

```
(defmacro read-with-echo ((stream object echoed &key prefix) &body body &aux
                          (out (gensym)) (echo (gensym)) (rewind (gensym))
                          (raw-output (gensym)) (length (gensym)))
  `(with-open-stream (,out (make-string-output-stream))
     (with-open-stream (,echo (make-echo-stream ,stream ,out))
       (with-open-stream (,rewind (make-concatenated-stream
                                   ,@(when prefix
                                       `((make-string-input-stream ,prefix)))
                                   ,echo))
         (let* ((,object (read-preserving-whitespace ,rewind))
                (,raw-output (get-output-stream-string ,out))
                (,length (length ,raw-output))
                (,echoed (subseq ,raw-output
                                 0
                                 (if (or (eof-p (peek-char nil ,rewind nil *eof*))
                                         (token-delimiter-p (elt ,raw-output (1- ,length))))
                                     ,length
                                     (1- ,length)))))
           (declare (ignorable ,object ,echoed))
           ,@body)))))
```

56. Next, we define a class of objects called *markers* that denote abstract objects in source code. Some of these objects, such as newlines and comments, are ones that would ordinarily be ignored by the reader. Others, such as `()` and `'`, are indistinguishable after reading from other, semantically equivalent objects (here, *nil* and *quote*), but we want to preserve the distinction in the output. In fact, nearly every standard macro character in Common Lisp is 'lossy', in the sense that the text of the original source code can not be reliably recovered from the object returned by *read*.

But during weaving, we want to more closely approximate the original source code than would be possible using the standard reader. Markers are our solution to this problem: we define reader macro functions for all of the standard macro characters that return markers that let us reconstruct, to varying degrees of accuracy, what was originally given in the source.

If a marker is *bound*—i.e., if *marker-boundp* returns non-nil when called with it as an argument—then the tangler will call *marker-value* to obtain the associated value. (The weaver will never ask for a marker's value). Otherwise, the marker will be silently dropped from its containing form; this is used, e.g., for newlines and comments. The value need not be stored in the *value* slot, but often is.

```
(defclass marker ()
  ((value :reader marker-value :initarg :value)))
(defun markerp (x) (typep x 'marker))

(defgeneric marker-boundp (marker))
(defmethod marker-boundp ((marker marker))
  (slot-boundp marker 'value))
```

57. We'll provide *print-object* methods for all of our marker classes. These methods are distinct from the pretty-printing routines used by the weaver, and usually less precise, in that they don't try to approximate the original source form. The idea of these methods is to produce a printed representation of an object that is semantically equivalent to the one originally specified.

We'll also define also a global variable, **print-marker**, that controls the way markers are printed. If it is true (as it is by default), then markers will be printed as just described. If it is false, markers are printed using the unreadable '`#<`' notation. This can be useful for debugging some of the reader routines, but might break others, so be careful. Routines that depend on this being set should explicitly bind it.

```
⟨ Global variables 12 ⟩ +≡
(defvar *print-marker* t)
```

58. The simple method defined here suffices for many marker types: it simply prints the marker's value if it is bound. Markers that require specialized printing will override this method.

```
(defmethod print-object ((obj marker) stream)
   (if *print-marker*
      (when (marker-boundp obj)
         (write (marker-value obj) :stream stream))
      (print-unreadable-object (obj stream :type t :identity t)
         (when (marker-boundp obj)
            (princ (marker-value obj) stream)))))
```

59. A few of the markers behave differently when tangling for the purposes of evaluation (e.g., within a call to *load-web*) than when writing out a tangled Lisp source file. We need this distinction only for read-time evaluated constructs, such as `#.` and `#+`/`#-`.

```
⟨ Global variables 12 ⟩ +≡
(defvar *evaluating* nil)
```

60. Our first marker is for newlines, which we preserve for the purposes of indentation. They are represented in code forms by an unbound marker, so the tangler will ignore them.

Note that we don't set a macro character for `#\Newline` in inner-Lisp mode, since indentation is completely ignored there.

```
(defclass newline-marker (marker)
   ((indentation :accessor indentation :initform nil)))
(defun newlinep (obj) (typep obj 'newline-marker))
(set-macro-character #\Newline
                        (λ (stream char)
                           (declare (ignore stream char))
                           (make-instance 'newline-marker))
                        nil (readtable-for-mode :lisp))
```

61. The rest of the reader macro functions for standard macro characters are defined in the order given in section 2.4 of the ANSI Common Lisp standard. We override all of the standard macro characters except `#\)` and `#\"` (the former because the standard reader macro function just signals an error, which is fine, and the latter because we don't need markers for strings).

62. *Left-Parenthesis.* We have two different kinds of markers for lists. The first is one for empty lists, so that we can maintain a distinction that the standard Lisp reader does not: between the empty list and *nil*. The second, for non-empty lists, stores not just the elements of the list, but their character positions as well; this is what allows us to do our indentation tracking.

Note that we bind our empty-list marker to the value *'()* so that it's preserved during tangling.

```
(defclass empty-list-marker (marker) () (:default-initargs :value '()))
(defvar *empty-list* (make-instance 'empty-list-marker))

(defclass list-marker (marker)
   ((length :accessor list-marker-length :initarg :length)
    (list :accessor list-marker-list :initarg :list)
    (charpos :accessor list-marker-charpos :initarg :charpos)))
(defun list-marker-p (obj) (typep obj 'list-marker))

(defclass consing-dot-marker (marker) ())
(defvar *consing-dot* (make-instance 'consing-dot-marker))

(defmethod marker-boundp ((marker list-marker)) t)
(defmethod marker-value ((marker list-marker))
   (do* ((list (list nil))
         (tail list)
         (marker-list (list-marker-list marker) (cdr marker-list))
         (x (car marker-list) (car marker-list)))
        ((endp marker-list) (cdr list))
      (cond ((eq x *consing-dot*)
               (rplacd tail ⟨Find the tail of the list marker 63⟩)
               (return (cdr list)))
             ((markerp x)
               (when (marker-boundp x)
                  (let ((obj (list x)))
                     (rplacd tail obj)
                     (setq tail obj))))
             (t (let ((obj (list x)))
                   (rplacd tail obj)
                   (setq tail obj))))))
```

63. There might be more than one object in a list marker following a consing dot, because of unbound markers (e.g., newlines and comments). So we just use the first bound marker or non-marker object that we find.

⟨Find the tail of the list marker 63⟩ ≡

```
(dolist (x marker-list (error "Nothing␣after␣.␣in␣list"))
   (when (or (not (markerp x))
             (and (markerp x)
                  (marker-boundp x)))
      (return x)))
```

This code is used in section 62.

64. We don't use *list-marker*s at all in inner-Lisp mode (since we don't do indentation tracking there), but we still want markers for the empty list. The function *make-list-reader* returns a closure that peeks ahead in the given stream looking for a closing parenthesis, and either returns an empty-list marker or invokes a full list-reading function, which is stored in the variable *next*. For inner-Lisp mode, that function is the standard reader macro function for '#\('.

```
(defun make-list-reader (next)
   (λ (stream char)
      (if (char= (peek-char t stream t nil t) #\) )
         (progn (read-char stream t nil t) *empty-list*)
         (funcall next stream char))))
(set-macro-character #\( (make-list-reader (get-macro-character #\( nil))
                        nil (readtable-for-mode :inner-lisp))
```

65. In Lisp mode, we need a full list reader that records character positions of the list elements. This would be almost straightforward if not for the consing dot.

```
(defun list-reader (stream char)
   (declare (ignore char))
   (loop with list = '()
         with charpos-list = '()
         for n upfrom 0
         and next-char = (peek-char t stream t nil t)
         as charpos = (stream-charpos stream)
         until (char= #\) next-char)
         if (char= #\. next-char)
            do ⟨Read the next token from stream, which might be a consing dot 66⟩
         else
            do ⟨Read the next object from stream and push it onto list 67⟩
         finally (read-char stream t nil t)
                 (return (make-instance 'list-marker
                                         :length n
                                         :list (nreverse list)
                                         :charpos (nreverse charpos-list)))))
(set-macro-character #\( (make-list-reader #'list-reader)
                        nil (readtable-for-mode :lisp))
```

66. If the next character is a dot, it could either be a consing dot, or the beginning of a token that happens to start with a dot. We decide by looking at the character *after* the dot: if it's a delimiter, then it *was* a consing dot; otherwise, we rewind and carefully read in the next object.

```
⟨Read the next token from stream, which might be a consing dot 66⟩ ≡
(with-rewind-stream (stream stream)
   (read-char stream t nil t) ; consume dot
   (let ((following-char (read-char stream t nil t)))
      (cond ((token-delimiter-p following-char)
               (unless (or list *read-suppress*)
                  (simple-reader-error stream "Nothing␣appears␣before␣.␣in␣list."))
               (push *consing-dot* list)
               (push charpos charpos-list))
              (t (rewind)
                 ⟨Read the next object from stream and push it onto list 67⟩)))))
```

This code is used in section 65.

67. We have to be careful when reading in a list, because the next character might be a macro character whose associated reader macro function returns zero values, and we don't want to accidentally read the closing '#\)'.

⟨ Read the next object from *stream* and push it onto *list* 67 ⟩ ≡
```
(let ((values (read-maybe-nothing stream t nil t)))
   (when values
      (push (car values) list)
      (push charpos charpos-list)))
```
This code is used in sections 65 and 66.

68. *Single-Quote.* We want to distinguish between a form quoted with a single-quote and one quoted (for whatever reason) with *quote*, another distinction ignored by the standard Lisp reader. We'll use this marker class for `#'`, too, which is why it's a little more general than one might think is needed.

```
(defclass quote-marker (marker)
   ((quote :reader quote-marker-quote :initarg :quote)
    (form :reader quoted-form :initarg :form)))
(defmethod marker-boundp ((marker quote-marker)) t)
(defmethod marker-value ((marker quote-marker))
   (list (quote-marker-quote marker) (quoted-form marker)))
(defun single-quote-reader (stream char)
   (declare (ignore char))
   (make-instance 'quote-marker :quote 'quote :form (read stream t nil t)))
(dolist (mode '(:lisp :inner-lisp))
   (set-macro-character #\' #'single-quote-reader nil (readtable-for-mode mode)))
```

69. *Semicolon.* Comments in Lisp code also need to be preserved for output during weaving. Comment markers are always unbound, and are therefore stripped during tangling.

```
(defclass comment-marker (marker)
   ((text :reader comment-text :initarg :text)))
```

70. To read a comment, we accumulate all of the characters starting with the semicolon and ending just before the next newline, which we leave for the newline reader to pick up. If the comment is empty, though, the newline is consumed, and we return zero values. This provides for 'soft newlines'; i.e., line breaks in the source file that will not appear in the woven output.

```
(defun comment-reader (stream char)
   (if (char= (peek-char nil stream nil #\Newline t) #\Newline)
      (progn (read-char stream t nil t) (values))
      (make-instance 'comment-marker
                     :text ⟨ Read characters up to, but not including, the next newline 71 ⟩)))
(set-macro-character #\; #'comment-reader nil (readtable-for-mode :lisp))
```

71. ⟨ Read characters up to, but not including, the next newline 71 ⟩ ≡
```
(with-output-to-string (s)
   (write-char char s) ; include the opening #\;
   (do ()
       ((char= (peek-char nil stream nil #\Newline t) #\Newline))
      (write-char (read-char stream t nil t) s)))
```
This code is used in section 70.

72. *Backquote* is hairy, and so we use a kludge to avoid implementing the whole thing. Our reader macro functions for #\` and #\, do the absolute minimum amount of processing necessary to be able to reconstruct the forms that were read. When the tangler asks for the *marker-value* of a backquote marker, we reconstruct the source form using the printer, then read it back in. The read is done using inner-Lisp mode so that we can pick up named section references, and we'll bind the global variable ∗*re-reading*∗ to true to avoid infinite regress.

Of course, this trick assumes read-print equivalence, but that's not unreasonable, since without it the file tangler won't work anyway. It's also the reason we need *print-object* methods for all the markers.

```
(defclass backquote-marker (marker)
  ((form :reader backq-form :initarg :form)))
(defmethod marker-boundp ((marker backquote-marker)) t)
(defmethod marker-value ((marker backquote-marker))
  (let ((*print-pretty* nil)
        (*print-readably* t)
        (*print-marker* t)
        (*readtable* (readtable-for-mode :inner-lisp))
        (*re-reading* t))
    (values (read-from-string (prin1-to-string marker)))))
(defmethod print-object ((obj backquote-marker) stream)
  (if *print-marker*
      (format stream "`~W" (backq-form obj))
      (print-unreadable-object (obj stream :type t :identity t))))
(defun backquote-reader (stream char)
  (if *re-reading*
      (funcall (get-macro-character char (readtable-for-mode nil)) stream char)
      (make-instance 'backquote-marker :form (read stream t nil t))))
(dolist (mode '(:lisp :inner-lisp))
  (set-macro-character #\` #'backquote-reader nil (readtable-for-mode mode)))
```

73. ⟨ Global variables 12 ⟩ +≡

```
(defvar *re-reading* nil)
```

74. In order to support named section references in backquoted forms, we need to be able to print them readably. The simple and obvious method works fine, since 'readably' here means readable in inner-Lisp mode.

```
(defmethod print-object ((obj named-section) stream)
  (format stream "@<~A@>" (section-name obj)))
```

75. *Comma* is really just part of the backquote-syntax, and so we're after the same goal as above: reconstructing just enough of the original source so that the reader can do what it would have done had we not been here in the first place.

Note that comma markers are bound, but self-evaluating: they need to be printed and re-read as part of a backquote form to retrieve their actual value.

```
(defclass comma-marker (marker)
  ((form :reader comma-form :initarg :form)
   (modifier :reader comma-modifier :initarg :modifier))
  (:default-initargs :modifier nil))
(defmethod marker-boundp ((marker comma-marker)) t)
(defmethod marker-value ((marker comma-marker)) marker)
(defmethod print-object ((obj comma-marker) stream)
  (if *print-marker*
    (format stream ",~@[~C~]~W" (comma-modifier obj) (comma-form obj))
    (print-unreadable-object (obj stream :type t :identity t))))
(defun comma-reader (stream char)
  (if *re-reading*
    (funcall (get-macro-character char (readtable-for-mode nil)) stream char)
    (case (peek-char nil stream t nil t)
      ((#\@ #\.) (make-instance 'comma-marker
                                :modifier (read-char stream)
                                :form (read stream t nil t)))
      (t (make-instance 'comma-marker :form (read stream t nil t))))))
(dolist (mode '(:lisp :inner-lisp))
  (set-macro-character #\, #'comma-reader nil (readtable-for-mode mode)))
```

76. Allegro Common Lisp's pretty printer tries to be clever about certain forms, like *defun* and *cond*, which might have a list as their *cadr*. However, it fails to notice when such a list is one constructed by their reader for a comma. It therefore doesn't print such lists using the comma syntax, yielding forms that can't be read in other Lisps. The following pretty printing routines work around this problem.

```
(defun pprint-list (stream list)
  (format stream "~:@<~/pprint-fill/~:>" list))
(defun pprint-comma (stream list)
  (format stream "~[,~;,@~;,.~]~W"
         (position (car list)
                   #+:allegro '(excl::bq-comma excl::bq-comma-atsign excl::bq-comma-dot)
                   #-:allegro '(:comma :comma-atsign :comma-dot))
         (cadr list)))
#+:allegro (deftype comma () '(member excl::bq-comma excl::bq-comma-atsign excl::bq-comma-dot))
#+:allegro (deftype broken-pprint-operators () '(member defun defmacro macrolet cond))
#+:allegro (set-pprint-dispatch '(cons comma) #'pprint-comma)
#+:allegro (set-pprint-dispatch '(cons broken-pprint-operators) #'pprint-list)
```

77. *Sharpsign* is the all-purpose dumping ground for Common Lisp reader macros. Because it's a dispatching macro character, we have to handle each sub-char individually, and unfortunately we need to override most of them. We'll handle them in the order given in section 2.4.8 of the CL standard.

78. Sharpsign single-quote is just like single-quote, except that the form is 'quoted' with *function* instead of *quote*.

```
(defclass function-marker (quote-marker) ())
(defun sharpsign-quote-reader (stream sub-char arg)
  (declare (ignore sub-char arg))
  (make-instance 'function-marker :quote 'function :form (read stream t nil t)))
(dolist (mode '(:lisp :inner-lisp))
  (set-dispatch-macro-character #\# #\' #'sharpsign-quote-reader (readtable-for-mode mode)))
```

79. Sharpsign left-parenthesis creates *simple-vector*s. The feature that we care about preserving is the length specification and consequent possible abbreviation.

```
(defclass simple-vector-marker (marker)
  ((length :initarg :length)
   (elements :initarg :elements)
   (element-type :initarg :element-type))
  (:default-initargs :element-type t))
(defmethod marker-boundp ((marker simple-vector-marker)) t)
(defmethod marker-value ((marker simple-vector-marker))
  (with-slots (elements element-type) marker
    (if (slot-boundp marker 'length)
      (with-slots (length) marker
        (let ((supplied-length (length elements)))
          (fill (replace (make-array length :element-type element-type) elements)
              (elt elements (1- supplied-length))
              :start supplied-length)))
      (coerce elements `(vector ,element-type)))))
;; Adapted from SBCL's sharp-left-paren.
(defun simple-vector-reader (stream sub-char arg)
  (declare (ignore sub-char))
  (let* ((list (read-delimited-list #\) stream t))
      (length (handler-case (length list)
                (type-error (error)
                  (declare (ignore error))
                  (simple-reader-error stream "improper␣list␣in␣#():␣~S" list)))))
    (unless *read-suppress*
      (if arg
        (if (> length arg)
          (simple-reader-error stream "vector␣longer␣than␣specified␣length:␣#~S~S" arg list)
          (make-instance 'simple-vector-marker :length arg :elements list))
        (make-instance 'simple-vector-marker :elements list)))))
(dolist (mode '(:lisp :inner-lisp))
  (set-dispatch-macro-character #\# #\( #'simple-vector-reader (readtable-for-mode mode)))
```

80. Sharpsign asterisk is similar, but the token following the asterisk must be composed entirely of `0`s and `1`s, from which a *simple-bit-vector* is constructed. It supports the same kind of abbreviation that `#()` does.

Note the use of the word 'token' above. By defining `#*` in terms of the *token* following the `*`, the authors of the standard have made it very, very difficult for a portable program to emulate the specified behavior, since only the built-in reader knows how to tokenize for the current readtable. What we do is resort to a dirty trick: we set up an echo stream, use the standard reader to parse the bit vector, then build our marker from the echoed characters.

```
(defclass bit-vector-marker (simple-vector-marker) ()
  (:default-initargs :element-type 'bit))
(defun simple-bit-vector-reader (stream sub-char arg)
  (declare (ignore sub-char))
  (let ((*readtable* (readtable-for-mode nil)))
    (read-with-echo (stream vector bits :prefix (format nil "#~@[~D~]*" arg))
      (apply #'make-instance 'bit-vector-marker
             :elements ⟨Build a bit vector from the characters in bits 81⟩
             (if arg (list :length arg))))))
(dolist (mode '(:lisp :inner-lisp))
  (set-dispatch-macro-character #\# #\* #'simple-bit-vector-reader (readtable-for-mode mode)))
```

81. The string *bits* now contains the '0' and '1' characters that make up the bit vector. But it might also contain the delimiter that terminates the token, so we have to be careful.

```
⟨Build a bit vector from the characters in bits 81⟩ ≡
(map 'bit-vector
     (λ (c) (ecase c (#\0 0) (#\1 1)))
     (subseq bits 0 (let ((n (length bits)))
                      (case (elt bits (1- n)) ((#\0 #\1) n) (t (1- n))))))
```

This code is used in section 80.

82. Sharpsign dot permits read-time evaluation. Ordinarily, of course, the form evaluated is lost, as only the result of the evaluation is returned. We want to preserve the form for both weaving and tangling to a file. But we also want to return the evaluated form as the *marker-value* when we're tangling for evaluation. So if we're not evaluating, we return a special 'pseudo-marker' with a specialized *print-object* method. This gives us an appropriate value in all three situations: during weaving, we have just another marker; when tangling for evaluation, we get the read-time-evaluated value; and in a tangled source file, we get a `#.` form.

```
(defclass read-time-eval ()
  ((form :reader read-time-eval-form :initarg :form)))
(defmethod print-object ((obj read-time-eval) stream)
  (if *print-marker*
      (format stream "#.~W" (read-time-eval-form obj))
      (print-unreadable-object (obj stream :type t :identity t))))
(defclass read-time-eval-marker (read-time-eval marker) ())
(defmethod marker-boundp ((marker read-time-eval-marker)) t)
(defmethod marker-value ((marker read-time-eval-marker))
  (if *evaluating*
      (call-next-method)
      (make-instance 'read-time-eval :form (read-time-eval-form marker))))
(defun sharpsign-dot-reader (stream sub-char arg)
  (declare (ignore sub-char arg))
  (let* ((*readtable* (if *evaluating* (readtable-for-mode nil) *readtable*))
         (form (read stream t nil t)))
    (unless *read-suppress*
      (unless *read-eval*
        (simple-reader-error stream "can't␣read␣#.␣while␣*READ-EVAL*␣is␣NIL"))
      (make-instance 'read-time-eval-marker :form form :value (eval form)))))
(dolist (mode '(:lisp :inner-lisp))
  (set-dispatch-macro-character #\# #\. #'sharpsign-dot-reader (readtable-for-mode mode)))
```

83. Sharpsign B, O, X, and R all represent rational numbers in a specific radix, but the radix is discarded by the standard reader, which just returns the number. We use the standard reader macro function for getting the actual value, and store the radix in our marker.

```
(defclass radix-marker (marker)
  ((base :reader radix-marker-base :initarg :base)))
(defparameter *radix-prefix-alist* '((#\B . 2) (#\O . 8) (#\X . 16) (#\R . nil)))
(defun radix-reader (stream sub-char arg)
  (make-instance 'radix-marker
                 :base (or (cdr (assoc (char-upcase sub-char) *radix-prefix-alist*)) arg)
                 :value ⟨Call the standard reader macro function for #⟨sub-char⟩ 84⟩))
(dolist (mode '(:lisp :inner-lisp))
  (dolist (sub-char '(#\B #\b #\O #\o #\X #\x #\R #\r))
    (set-dispatch-macro-character #\# sub-char #'radix-reader (readtable-for-mode mode))))
```

84. ⟨Call the standard reader macro function for #⟨*sub-char*⟩ 84⟩ ≡

```
(funcall (get-dispatch-macro-character #\# sub-char (readtable-for-mode nil))
         stream sub-char arg)
```

This code is used in section 83.

85. Sharpsign S requires determining the standard constructor function of the structure type named, which we simply can't do portably. So we use the same trick we used for backquote above: we cache the form as given, then dump it out to a string and let the standard reader parse it when we need the value.

```
(defclass structure-marker (marker)
  ((form :reader structure-marker-form :initarg :form)))
(defmethod marker-boundp ((marker structure-marker)) t)
(defmethod marker-value ((marker structure-marker))
  (let ((*print-pretty* nil)
        (*print-readably* t)
        (*print-marker* t)
        (*readtable* (readtable-for-mode nil)))
    (values (read-from-string (prin1-to-string marker)))))
(defmethod print-object ((obj structure-marker) stream)
  (if *print-marker*
    (format stream "#S~W" (structure-marker-form obj))
    (print-unreadable-object (obj stream :type t :identity t))))
(defun structure-reader (stream sub-char arg)
  (declare (ignore sub-char arg))
  (make-instance 'structure-marker :form (read stream t nil t)))
(dolist (mode '(:lisp :inner-lisp))
  (set-dispatch-macro-character #\# #\S #'structure-reader (readtable-for-mode mode)))
```

86. Sharpsign + and − provide read-time conditionalization based on feature expressions, described in section 24.1.2 of the CL standard. This routine, adapted from SBCL, interprets such an expression.

```
(defun featurep (x)
  (etypecase x
    (cons
     (case (car x)
       ((:not not)
        (cond
          ((cddr x) (error "too␣many␣subexpressions␣in␣feature␣expression:␣~S" x))
          ((null (cdr x)) (error "too␣few␣subexpressions␣in␣feature␣expression:␣~S" x))
          (t (not (featurep (cadr x))))))
       ((:and and) (every #'featurep (cdr x)))
       ((:or or) (some #'featurep (cdr x)))
       (t
        (error "unknown␣operator␣in␣feature␣expression:␣~S." x))))
    (symbol (not (null (member x *features* :test #'eq))))))
```

87. For sharpsign +/−, we use the same sort of trick we used for `#.` above: we have a marker that returns the appropriate value when tangling for evaluation, but returns a 'pseudo-marker' when tangling to a file, so that we can preserve the original form.

This case is slightly trickier, though, because we can't just call *read* on the form, since if the the test fails, ∗*read-suppress*∗ will be true, and we won't get anything back. So we use an echo stream to catch the raw characters that the reader scans, and use that to reconstruct the form.

```
(defclass read-time-conditional ()
  ((plusp :reader read-time-conditional-plusp :initarg :plusp)
   (test :reader read-time-conditional-test :initarg :test)
   (form :reader read-time-conditional-form :initarg :form)))
(defmethod print-object ((obj read-time-conditional) stream)
  (if *print-marker*
    (format stream "#~:[-~;+~]~S␣~A"
            (read-time-conditional-plusp obj)
            (read-time-conditional-test obj)
            (read-time-conditional-form obj))
    (print-unreadable-object (obj stream :type t :identity t))))
(defclass read-time-conditional-marker (read-time-conditional marker) ())
(defmethod marker-boundp ((marker read-time-conditional-marker))
  (if *evaluating*
    (call-next-method)
    t))
(defmethod marker-value ((marker read-time-conditional-marker))
  (if *evaluating*
    (call-next-method)
    (make-instance 'read-time-conditional
                   :plusp (read-time-conditional-plusp marker)
                   :test (read-time-conditional-test marker)
                   :form (read-time-conditional-form marker))))
(defun read-time-conditional-reader (stream sub-char arg)
  (declare (ignore arg))
  (let* ((plusp (ecase sub-char (#\+ t) (#\- nil)))
         (*readtable* (readtable-for-mode nil))
         (test (let ((*package* (find-package "KEYWORD"))
                     (*read-suppress* nil))
                 (read stream t nil t)))
         (*read-suppress* (if plusp (not (featurep test)) (featurep test))))
    (peek-char t stream t nil t)
    (read-with-echo (stream value form)
      (apply #'make-instance 'read-time-conditional-marker
             :plusp plusp :test test :form form
             (and (not *read-suppress*) (list :value value))))))
(dolist (mode '(:lisp :inner-lisp))
  (set-dispatch-macro-character #\# #\+ #'read-time-conditional-reader (readtable-for-mode mode))
  (set-dispatch-macro-character #\# #\- #'read-time-conditional-reader (readtable-for-mode mode)))
```

88. So much for the standard macro characters. Now we're ready to move on to WEB-specific reading. We accumulate TEX mode material such as commentary, section names, *&c.* using the following function, which reads from *stream* until encountering either EOF or an element of the *control-chars* argument, which should be a designator for a list of characters.

```
(defun snarf-until-control-char (stream control-chars &aux (control-chars (ensure-list control-chars)))
  (with-output-to-string (string)
    (loop for char = (peek-char nil stream nil *eof* nil)
          until (or (eof-p char) (member char control-chars))
          do (write-char (read-char stream) string))))
```

89. In TEX mode (including restricted contexts), we allow embedded Lisp code to be surrounded by |...|, where it is read in inner-Lisp mode.

```
(defun read-inner-lisp (stream char)
  (with-mode :inner-lisp
    (read-delimited-list char stream)))
(dolist (mode '(:tex :restricted))
  (set-macro-character #\| #'read-inner-lisp nil (readtable-for-mode mode)))
```

90. The call to *read-delimited-list* in *read-inner-lisp* will only stop at the closing | if we make it a terminating macro character, overriding its usual Lisp meaning as an escape character.

```
(set-macro-character #\| (get-macro-character #\) nil) nil (readtable-for-mode :inner-lisp))
```

91. We make #\@ a non-terminating dispatching macro character in every mode, and define some convenience routines for retrieving and setting the reader macro functions that implement the control codes.

```
(dolist (mode *modes*)
  ;; The CL standard does not say that calling make-dispatch-macro-character
  ;; on a character that's already a dispatching macro character is supposed
  ;; to signal an error, but SBCL does so anyway; hence the ignore-errors.
  (ignore-errors
    (make-dispatch-macro-character #\@ t (readtable-for-mode mode))))
(defun get-control-code (sub-char mode)
  (get-dispatch-macro-character #\@ sub-char (readtable-for-mode mode)))
(defun set-control-code (sub-char function &optional (modes *modes*))
  (dolist (mode (ensure-list modes))
    (set-dispatch-macro-character #\@ sub-char function (readtable-for-mode mode))))
```

92. The control code @@ yields the string "@" in all modes, but it should really only be used in TEX text.

```
(set-control-code #\@ (λ (stream sub-char arg)
                        (declare (ignore sub-char stream arg))
                        (string "@")))
```

93. Non-test sections are introduced by `@␣` or `@*`, which differ only in the way they are output during weaving. The reader macro functions that implement these control codes return an instance of the appropriate section class.

```
(defun start-section-reader (stream sub-char arg)
  (declare (ignore stream arg))
  (make-instance (ecase sub-char
                   (#\␣ 'section)
                   (#\* 'starred-section))))
(dolist (sub-char '(#\␣ #\*))
  (set-control-code sub-char #'start-section-reader '(:limbo :tex :lisp)))
```

94. Test sections are handled similarly, but are introduced with `@t`. Test sections may also be 'starred'. Immediately following whitespace is ignored.

```
(defun start-test-section-reader (stream sub-char arg)
  (declare (ignore sub-char arg))
  (prog1 (if (and (char= (peek-char t stream t nil t) #\*)
                  (read-char stream t nil t))
             (make-instance 'starred-test-section)
             (make-instance 'test-section))
         (loop until (char/= (peek-char t stream t nil t) #\Newline)
               do (read-char stream t nil t))))
(set-control-code #\t #'start-test-section-reader '(:limbo :tex :lisp))
```

95. The control codes `@l` and `@p` (where 'l' is for 'Lisp' and 'p' is for 'program'—both control codes do the same thing) begin the code part of an unnamed section. They are valid only in TEX mode; every section must begin with a commentary, even if it is empty. We set the control codes in Lisp mode only for error detection.

```
(defclass start-code-marker (marker)
  ((name :reader section-name :initarg :name))
  (:default-initargs :name nil))
(defun start-code-reader (stream sub-char arg)
  (declare (ignore stream sub-char arg))
  (make-instance 'start-code-marker))
(dolist (sub-char '(#\l #\p))
  (set-control-code sub-char #'start-code-reader '(:tex :lisp)))
```

96. The control code `@e` ('e' for 'eval') indicates that the following form should be evaluated by the section reader, *in addition to* being tangled into the output file. Evaluated forms should be used only for establishing state that is needed by the reader: package definitions, structure definitions that are used with `#S`, *&c.*

```
(defclass evaluated-form-marker (marker) ())
(defun read-evaluated-form (stream sub-char arg)
  (declare (ignore sub-char arg))
  (loop for form = (read stream t nil t)
        until (not (newlinep form)) ; skip over newlines
        finally (return (make-instance 'evaluated-form-marker :value form))))
(set-control-code #\e #'read-evaluated-form :lisp)
```

97. Several control codes, including `@<`, contain 'restricted' TeX text, called *control text*, that extends to the next `@>`. When we first read control text, we ignore inner-Lisp material (that is, Lisp forms surrounded by `|...|`). During weaving, we'll re-scan it to pick up such material. The reason for this two-pass approach is that control text will frequently be used as keys in the various binary search trees, so it's convenient to keep it in string form until the last possible moment (i.e., right up until we need to print it out).

The only control codes that are valid in restricted mode are `@>` and `@@`.

```
(defvar *end-control-text* (make-symbol "@>"))
(set-control-code #\> (constantly *end-control-text*) :restricted)
(defun read-control-text (stream &optional (eof-error-p t) eof-value recursive-p)
  (with-output-to-string (string)
    (with-mode :restricted
      (loop for text = (snarf-until-control-char stream #\@)
            for next = (read-preserving-whitespace stream eof-error-p eof-value recursive-p)
            do (write-string text string)
            if (eq next *end-control-text*) do (loop-finish)
            else do (write-string next string)))))
```

98. The control code `@<` introduces a section name, which extends to the closing `@>`. Its meaning is context-dependent.

In TeX mode, a name must be followed by `=`, which attaches the name to the current section and begins the code part.

In Lisp and inner-Lisp modes, a name is taken to refer to the section so named. During tangling, such references in Lisp mode will be replaced with the code defined for that section. References in inner-Lisp mode are only citations, and so are not expanded.

```
(defun make-section-name-reader (definition-allowed-p use)
  (λ (stream sub-char arg)
    (declare (ignore sub-char arg))
    (let* ((name (read-control-text stream t nil t))
           (definitionp (eql (peek-char nil stream nil nil t) #\=)))
      (if definitionp
        (if definition-allowed-p
          (progn
            (read-char stream)
            (make-instance 'start-code-marker :name name))
          ⟨Signal an error about section definition in Lisp mode 100⟩)
        (if definition-allowed-p
            ⟨Signal an error about section name use in TeX mode 101⟩
            (let ((named-section (find-section name)))
              (if use
                (pushnew *current-section* (used-by named-section))
                (pushnew *current-section* (cited-by named-section)))
              named-section))))))
(set-control-code #\< (make-section-name-reader t nil) :tex)
(set-control-code #\< (make-section-name-reader nil t) :lisp)
(set-control-code #\< (make-section-name-reader nil nil) :inner-lisp)
```

99. ⟨ Condition classes 27 ⟩ +≡

```
(define-condition section-name-context-error (error)
  ((name :reader section-name :initarg :name)))
(define-condition section-name-definition-error (section-name-context-error)
  ()
  (:report (λ (condition stream)
             (format stream "Can't␣define␣a␣named␣section␣in␣Lisp␣mode:␣~A"
                     (section-name condition)))))
(define-condition section-name-use-error (section-name-context-error)
  ()
  (:report (λ (condition stream)
             (format stream "Can't␣use␣a␣section␣name␣in␣TeX␣mode:␣~A"
                     (section-name condition)))))
```

100. ⟨ Signal an error about section definition in Lisp mode 100 ⟩ ≡

```
(restart-case (error 'section-name-definition-error :name name)
  (use-section ()
    :report "Don't␣define␣the␣section,␣just␣use␣it."
    (find-section name)))
```

This code is used in section 98.

101. ⟨ Signal an error about section name use in TEX mode 101 ⟩ ≡

```
(restart-case (error 'section-name-use-error :name name)
  (name-section ()
    :report "Name␣the␣current␣section␣and␣start␣the␣code␣part."
    (make-instance 'start-code-marker :name name))
  (cite-section ()
    :report "Assume␣the␣section␣is␣just␣being␣cited."
    (find-section name)))
```

This code is used in section 98.

102. The control code `@x` reads the following form, which should be a designator for a list of packages, and informs the indexing sub-system that symbols in those packages are to be indexed. It returns the form.

The usual idiom is to use this control code in a *defpackage* form; e.g., *(defpackage @x*`"FOO"` . . .*)* will index the symbols in the package named `FOO`.

Note that this is *completely different* than the `@x` control code of `WEB` and `CWEB`, which is part of their change-file system.

```
(defun index-package-reader (stream sub-char arg)
  (declare (ignore sub-char arg))
  (let ((form (read stream)))
    (index-package form)
    form))
(set-control-code #\x #'index-package-reader :lisp)
```

103. These next control codes are used to manually create index entries. They differ only in how they are typeset in the woven output.

```
(defun index-entry-reader (stream sub-char arg)
  (declare (ignore arg))
  (add-index-entry *index*
                   (make-instance (ecase sub-char
                                    (#\^ 'heading)
                                    (#\. 'tt-heading)
                                    (#\: 'custom-heading))
                                  :name (read-control-text stream))
                   *current-section*)
  (values))
(dolist (sub-char '(#\^ #\. #\:))
  (set-control-code sub-char #'index-entry-reader '(:tex :lisp)))
```

104. When we're accumulating forms from the code part of a section, we'll interpret two newlines in a row as ending a paragraph, as in TeX.

```
(defclass par-marker (newline-marker) ())
(defvar *par* (make-instance 'par-marker))
```

105. We need one last utility before coming to the main section reader. When we're accumulating text, we don't want to bother with empty strings. So we use the following macro, which is like *push*, but does nothing if the new object is an empty string or *nil*.

```
(defmacro maybe-push (obj place &aux (g (gensym)))
  `(let ((,g ,obj))
     (when (if (stringp ,g) (plusp (length ,g)) ,g)
       (push ,g ,place))))
```

106. We come now to the heart of the `WEB` parser. This function is a tiny state machine that models the global syntax of a `WEB` file. (We can't just use reader macros since sections and their parts lack explicit closing delimiters.) It returns a list of *section* objects.

```
(defun read-sections (input-stream &key (append t))
  (with-charpos-input-stream (stream input-stream)
    (flet ((finish-section (section commentary code)
             ⟨ Trim whitespace and reverse commentary and code 114 ⟩
             (setf (section-commentary section) commentary)
             (setf (section-code section) code)
             (when (section-name section)
               (let ((named-section (find-section (section-name section))))
                 (if append
                     (push section (named-section-sections named-section))
                     (setf (named-section-sections named-section) (list section)))))
             section))
      (prog (form commentary code section sections)
         (setq section (make-instance 'limbo-section))
         ⟨ Accumulate limbo text in commentary 107 ⟩
       commentary
         ⟨ Finish the last section and initialize section variables 108 ⟩
         ⟨ Accumulate TEX-mode material in commentary 109 ⟩
       lisp
         ⟨ Accumulate Lisp-mode material in code 110 ⟩
       eof
         (push (finish-section section commentary code) sections)
         (return (nreverse sections))))))
```

107. Limbo text is TEX text that proceeds the first section marker, and we treat it as commentary for a special section with no code. Note that inner-Lisp material is not recognized in limbo text.

```
⟨ Accumulate limbo text in commentary 107 ⟩ ≡
(with-mode :limbo
  (loop
    (maybe-push (snarf-until-control-char stream #\@) commentary)
    (let ((next-input (read-maybe-nothing-preserving-whitespace stream nil *eof* nil)))
      (when next-input
        (typecase (setq form (car next-input))
          (eof (go eof))
          (section (go commentary))
          (t (push form commentary)))))))
```

This code is used in section 106.

108. ⟨ Finish the last section and initialize section variables 108 ⟩ ≡

```
(push (finish-section section commentary code) sections)
(check-type form section)
(setq section form
      commentary '()
      code '())
```

This code is used in sections 106 and 111.

109. The commentary part that begins a section consists of TEX text and inner-Lisp material surrounded by `|...|`. It is terminated by either the start of a new section, the beginning of the code part, or EOF. If a code part is detected, we also set the name of the current section, which may be *nil*.

⟨ Accumulate TEX-mode material in *commentary* 109 ⟩ ≡

```
(with-mode :tex
  (loop
    (maybe-push (snarf-until-control-char stream '(#\@ #\|)) commentary)
    (let ((next-input (read-maybe-nothing-preserving-whitespace stream nil *eof* nil)))
      (when next-input
        (typecase (setq form (car next-input))
          (eof (go eof))
          (section (go commentary))
          (start-code-marker (setf (section-name section) (section-name form))
                             (go lisp))
          (t (push form commentary)))))))
```

This code is used in section 106.

110. The code part of a section consists of zero or more Lisp forms and is terminated by either EOF or the start of a new section. This is also where we evaluate `@e` forms.

⟨ Accumulate Lisp-mode material in *code* 110 ⟩ ≡

```
(check-type form start-code-marker)
(with-mode :lisp
  (loop
    (let ((next-input (read-maybe-nothing-preserving-whitespace stream nil *eof* nil)))
      (when next-input
        (typecase (setq form (car next-input))
          (eof (go eof))
          (section (go commentary))
          (start-code-marker ⟨ Complain about starting a section without a commentary part 111 ⟩)
          (newline-marker ⟨ Maybe push the newline marker 113 ⟩)
          (evaluated-form-marker (let ((form (marker-value form)))
                                   (let ((*evaluating* t)
                                         (*readtable* (readtable-for-mode nil)))
                                     (eval (tangle form)))
                                   (push form code)))
          (t (push form code)))))))
```

This code is used in section 106.

This code is cited in section 114.

111. ⟨ Complain about starting a section without a commentary part 111 ⟩ ≡

```
(cerror "Start␣a␣new␣unnamed␣section␣with␣no␣commentary."
        'section-lacks-commentary :stream stream)
(setq form (make-instance 'section))
⟨ Finish the last section and initialize section variables 108 ⟩
```

This code is used in section 110.

112. ⟨ Condition classes 27 ⟩ +≡

```
(define-condition section-lacks-commentary (parse-error)
  ((stream :initarg :stream :reader section-lacks-commentary-stream))
  (:report (λ (error stream)
             (let∗ ((input-stream
                     (do ((stream (section-lacks-commentary-stream error)))
                         (())
                       (typecase stream
                         (echo-stream
                          (setq stream (echo-stream-input-stream stream)))
                         (t (return stream)))))
                    (position (file-position input-stream))
                    (pathname (when (typep input-stream 'file-stream)
                                (pathname input-stream))))
               (format stream
                       "~@<Can't␣start␣a␣section␣with␣a␣code␣part␣~
                       ~:[~;~:*at␣position␣~D␣in␣file␣~A.~]~:@>"
                       position (or pathname input-stream))))))
```

113. We won't push a newline marker if no code has been accumulated yet, and we'll push a paragraph marker instead if there are two newlines in a row.

⟨ Maybe push the newline marker 113 ⟩ ≡

```
(unless (null code)
  (cond ((newlinep (car code))
         (pop code)
         (push ∗par∗ code))
        (t (push form code))))
```

This code is used in section 110.

114. We trim trailing whitespace from the last string in *commentary*, leading whitespace from the first, and any trailing newline marker from *code*. (Leading newlines are handled in ⟨ Accumulate Lisp-mode material in *code* 110 ⟩.)

⟨ Trim whitespace and reverse *commentary* and *code* 114 ⟩ ≡

```
(when (stringp (car commentary))
  (rplaca commentary (string-right-trim ∗whitespace∗ (car commentary))))
(setq commentary (nreverse commentary))
(when (stringp (car commentary))
  (rplaca commentary (string-left-trim ∗whitespace∗ (car commentary))))
(setq code (nreverse (member-if-not #'newlinep code)))
```

This code is used in section 106.

115. The tangler. Tangling involves recursively replacing each reference to a named section with the code accumulated for that section. The function *tangle-1* expands one level of such references, returning the possibly-expanded form and a boolean representing whether or not any expansions were actually performed.

Note that this is a splicing operation, like '`,@`', not a simple substitution, like normal Lisp macro expansion: if ⟨foo n⟩≡*(x y)*, then *(tangle-1 '(a* ⟨foo n⟩ *b))* → *(a x y b)*, *t*.

Tangling also replaces bound markers with their associated values, and removes unbound markers. If the keyword argument *expand-named-sections* is false, then this is in fact all that *tangle-1* does; we'll use this capability later in the indexing code.

```
(defun tangle-1 (form &key (expand-named-sections t))
  (flet ((tangle-1 (form)
           (tangle-1 form :expand-named-sections expand-named-sections)))
    (typecase form
      (marker (values (marker-value form) t))
      (atom (values form nil))
      ((cons named-section *)
       (multiple-value-bind (d cdr-expanded-p) (tangle-1 (cdr form))
         (if expand-named-sections
             (values (append (section-code (car form)) d) t)
             (values (cons (car form) d) cdr-expanded-p))))
      ((cons marker *)
       (values (if (marker-boundp (car form))
                   (cons (marker-value (car form)) (tangle-1 (cdr form)))
                   (tangle-1 (cdr form)))
               t))
      (t (multiple-value-bind (a car-expanded-p) (tangle-1 (car form))
           (multiple-value-bind (d cdr-expanded-p) (tangle-1 (cdr form))
             (values (if (and (eql a (car form))
                              (eql d (cdr form)))
                         form
                         (cons a d))
                     (or car-expanded-p cdr-expanded-p))))))))
```

116. *tangle* repeatedly calls *tangle-1* on *form* until it can no longer be expanded. Like *tangle-1*, it returns the possibly-expanded form and an 'expanded' flag. This code is adapted from SBCL's *macroexpand*.

```
(defun tangle (form &key (expand-named-sections t))
  (labels ((expand (form expanded)
             (multiple-value-bind (new-form newly-expanded-p)
                 (tangle-1 form
                           :expand-named-sections expand-named-sections)
               (if newly-expanded-p
                   (expand new-form t)
                   (values new-form expanded)))))
    (expand form nil)))
```

117. This little utility function returns a list of all of the forms in all of the unnamed sections' code parts. This is our first-order approximation of the complete program; if you tangle it, you get the whole thing.

```
(defun unnamed-section-code-parts (sections)
  (mapappend #'section-code (coerce (remove-if #'section-name sections) 'list)))
```

118. We're now ready for the high-level tangler interface. We begin with *load-web*, which uses a helper function, *load-web-from-stream*, so that it can handle input from an arbitrary stream. The logic is straightforward: we loop over the tangled forms read from the stream, evaluating each one in turn.

Note that like *load*, we bind ∗*readtable*∗ and ∗*package*∗ to their current values, so that assignments to those variables in the WEB code will not affect the calling environment.

```
(defun load-web-from-stream (stream print &key (append t) &aux
                                   (*readtable* *readtable*)
                                   (*package* *package*)
                                   (*evaluating* t))
  (dolist (form (tangle (unnamed-section-code-parts
                          (read-sections stream :append append))) t)
    (if print
      (let ((results (multiple-value-list (eval form))))
        (format t "~&;␣~{~S~^,␣~}~%" results))
      (eval form))))

(defun load-web (filespec &key
                 (verbose *load-verbose*)
                 (print *load-print*)
                 (if-does-not-exist t)
                 (external-format :default))
  "Load␣the␣web␣given␣by␣FILESPEC␣into␣the␣Lisp␣environment."
  ⟨ Initialize global variables 13 ⟩
  (when verbose (format t "~&;␣loading␣WEB␣from␣~S~%" filespec))
  (if (streamp filespec)
    (load-web-from-stream filespec print)
    (with-open-file (stream (merge-pathnames filespec (make-pathname :type "CLW" :case :common))
                     :direction :input
                     :external-format external-format
                     :if-does-not-exist (if if-does-not-exist :error nil))
      (load-web-from-stream stream print))))
```

119. This next function exists solely for the sake of front-ends that wish to load a piece of a WEB, such as the author's '`clweb.el`'. Note that it does *not* initialize the global variables like ∗*named-sections*∗; this allows for incremental redefinition.

```
(defun load-sections-from-temp-file (file append &aux
                                        (truename (probe-file file)))
  "Load␣web␣sections␣from␣FILE,␣then␣delete␣it."
  (when truename
    (unwind-protect
        (with-open-file (stream truename :direction :input)
          (load-web-from-stream stream t :append append))
      (delete-file truename))))
```

120. Both *tangle-file* and *weave*, below, take a *tests-file* argument that has slightly hairy defaulting behavior. If it's supplied and is non-*nil*, then we use a pathname derived from the one given by merging in a default type (`"lisp"` in the case of tangling, `"tex"` for weaving). If it's not supplied, then we construct a pathname from the output file by appending the string `"-tests"` to its name component. Finally, if the argument is supplied and is *nil*, then no tests file will be written at all.

```
(defun tests-file-pathname (output-file type &key (tests-file nil tests-file-supplied-p) &allow-other-keys)
  (if tests-file
    (merge-pathnames tests-file (make-pathname :type type :case :common))
    (unless tests-file-supplied-p
      (make-pathname :name (concatenate 'string
                                         (pathname-name output-file :case :common)
                                         "-TESTS")
                     :type type
                     :defaults output-file
                     :case :common))))
```

121. The file tangler operates by writing out the tangled code to a Lisp source file and then invoking the file compiler on that file. The arguments are essentially the same as those to *compile-file*, except for the *tests-file* keyword argument, which specifies the file to which the test sections' code should be written.

```
(defun tangle-file (input-file &rest args &key
                    output-file
                    tests-file
                    (verbose *compile-verbose*)
                    (print *compile-print*)
                    (external-format :default) &allow-other-keys &aux
                    ⟨ Merge defaults for tangler file names 122 ⟩
                    (*readtable* *readtable*)
                    (*package* *package*))
  "Tangle␣and␣compile␣the␣web␣in␣INPUT-FILE,␣producing␣OUTPUT-FILE."
  (declare (ignore output-file tests-file))
  (when verbose (format t "~&;␣tangling␣web␣from␣~A:~%" input-file))
  ⟨ Initialize global variables 13 ⟩
  (with-open-file (input input-file
                    :direction :input
                    :external-format external-format)
    (read-sections input))
  ⟨ Complain about any unused named sections 124 ⟩
  (flet ((write-forms (sections output-file)
           (with-open-file (output output-file
                            :direction :output
                            :if-exists :supersede
                            :external-format external-format)
             (format output ";;;;␣TANGLED␣WEB␣FROM␣\"~A\".␣DO␣NOT␣EDIT." input-file)
             (let ((*evaluating* nil)
                   (*print-marker* t)
                   (*print-level* nil))
               (dolist (form (tangle (unnamed-section-code-parts sections)))
                 (pprint form output))))))
    (when (and tests-file
               (> (length *test-sections*) 1)) ; there's always a limbo section
      (when verbose (format t "~&;␣writing␣tests␣to␣~A~%" tests-file))
      (write-forms *test-sections* tests-file)
      (compile-file tests-file ; use default output file
                    :verbose verbose
                    :print print
                    :external-format external-format))
    (when verbose (format t "~&;␣writing␣tangled␣code␣to␣~A~%" lisp-file))
    (write-forms *sections* lisp-file)
    (apply #'compile-file lisp-file :allow-other-keys t args)))
```

122. ⟨ Merge defaults for tangler file names 122 ⟩ ≡

```
(input-file (merge-pathnames input-file (make-pathname :type "CLW" :case :common)))
(output-file (apply #'compile-file-pathname input-file :allow-other-keys t args))
(lisp-file (merge-pathnames (make-pathname :type "LISP" :case :common) output-file))
(tests-file (apply #'tests-file-pathname output-file "LISP" args))
```

This code is used in section 121.

123. A named section doesn't do any good if it's never referenced, so we issue warnings about unused named sections.

⟨ Condition classes 27 ⟩ +≡
```
(define-condition unused-named-section-warning (simple-warning) ())
```

124. ⟨ Complain about any unused named sections 124 ⟩ ≡
```
(let ((unused-sections '()))
  (flet ((note-unused-section (section)
           (when (null (used-by section))
             (push section unused-sections))))
    (maptree #'note-unused-section *named-sections*)
    (map nil
         (λ (section)
           (warn 'unused-named-section-warning
                 :format-control "Unused␣named␣section␣<~A>."
                 :format-arguments (list (section-name section))))
         (sort unused-sections #'< :key #'section-number))))
```
This code is used in section 121.

125. The weaver. The top-level weaver interface is modeled after *compile-file*. The function *weave* reads the WEB *input-file* and produces an output TEX file named by *output-file*. If *output-file* is not supplied or is *nil*, a pathname will be generated from *input-file* by replacing its *type* component with "tex".

If *verbose* is true, *weave* prints a message in the form of a comment to standard output indicating what file is being woven. If *verbose* is not supplied, the value of **weave-verbose** is used.

If *print* is true, the section number of each section encountered is printed to standard output, with starred sections prefixed with *. If *print* is not supplied, the value of **weave-print** is used.

Finally, the *external-format* argument specifies the external file format to be used when opening both the input file and the output file. *N.B.:* standard TEX only handles 8-bit characters, and the encodings for non-printable-ASCII characters vary widely.

If successful, *weave* returns the truename of the output file.

```
(defun weave (input-file &rest args &key
                  output-file tests-file (index-file nil index-file-supplied-p)
                  (verbose *weave-verbose*)
                  (print *weave-print*)
                  (if-does-not-exist t)
                  (external-format :default) &aux
                  ⟨Merge defaults for weaver file names 126⟩
                  (*readtable* *readtable*)
                  (*package* *package*))
  "Weave␣the␣web␣contained␣in␣INPUT-FILE,␣producing␣the␣TeX␣file␣OUTPUT-FILE."
  (declare (ignore tests-file))
  (when verbose (format t "~&;␣weaving␣web␣from␣~A:~%" input-file))
  ⟨Initialize global variables 13⟩
  (with-open-file (input input-file
                        :direction :input
                        :external-format external-format
                        :if-does-not-exist (if if-does-not-exist :error nil))
    (read-sections input))
  (when (and tests-file
             (> (length *test-sections*) 1)) ; there's always a limbo section
    (when verbose (format t "~&;␣weaving␣tests␣to␣~A~%" tests-file))
    (weave-sections *test-sections*
                    :output-file tests-file
                    :print print
                    :verbose verbose
                    :external-format external-format))
   (when verbose (format t "~&;␣weaving␣sections␣to␣~A~%" output-file))
   (weave-sections *sections*
                   :output-file output-file
                   :index-file index-file
                   :sections-file sections-file
                   :verbose verbose
                   :print print
                   :external-format external-format))
```

126. ⟨ Merge defaults for weaver file names 126 ⟩ ≡

```
(input-file (merge-pathnames input-file (make-pathname :type "CLW" :case :common)))
(output-file (if output-file
                 (merge-pathnames output-file (make-pathname :type "TEX" :case :common))
                 (merge-pathnames (make-pathname :type "TEX" :case :common) input-file)))
(tests-file (apply #'tests-file-pathname output-file "TEX" args))
(index-file (if index-file
                (merge-pathnames index-file (make-pathname :type "IDX" :case :common))
                (when (not index-file-supplied-p)
                  (merge-pathnames (make-pathname :type "IDX" :case :common) output-file))))
(sections-file (when index-file
                 (merge-pathnames (make-pathname :type "SCN" :case :common) index-file)))
```

This code is used in section 125.

127. ⟨ Global variables 12 ⟩ +≡

```
(defvar *weave-verbose* t)
(defvar *weave-print* t)
```

128. The individual sections and their contents are printed using the pretty printer with a customized dispatch table.

⟨ Global variables 12 ⟩ +≡

```
(defparameter *weave-pprint-dispatch* (copy-pprint-dispatch nil))
```

129. The following routine does the actual writing of the sections to the output file.

```
(defun weave-sections (sections &key
                           output-file index-file sections-file
                           (verbose *weave-verbose*)
                           (print *weave-print*)
                           (external-format :default))
  (flet ((weave (object stream)
           (write object
                  :stream stream
                  :case :downcase
                  :escape nil
                  :level nil
                  :pretty t
                  :pprint-dispatch *weave-pprint-dispatch*
                  :right-margin 1000)))
    (macrolet ((with-output-file ((stream filespec) &body body)
                 `(with-open-file (,stream ,filespec
                                   :direction :output
                                   :external-format external-format
                                   :if-exists :supersede)
                    ,@body)))
      (with-output-file (out output-file)
        (format out "\\input␣clwebmac~%")
        (if print
          (pprint-logical-block (nil nil :per-line-prefix ";␣")
            (map nil
                 (λ (section)
                   (format t "~:[~;~:@_*~]~D~:_␣"
                             (typep section 'starred-section)
                             (section-number section))
                   (weave section out))
                 sections))
          (map nil (λ (section) (weave section out)) sections))
        (when index-file
          (when verbose (format t "~&;␣writing␣the␣index␣to␣~A~%" index-file))
          (with-output-file (idx index-file)
            (weave (index-sections sections) idx))
          (with-output-file (scn sections-file)
            (maptree (λ (section)
                       (weave (make-instance 'section-name-index-entry :named-section section) scn))
                     *named-sections*))
          (format out "~&\\inx~%\\fin~%\\con~%"))
        (format out "~&\\end~%")
        (truename out)))))
```

130. The rest of the weaver consists entirely of pretty-printing routines that are installed in **weave-pprint-dispatch**.

```
(defun set-weave-dispatch (type-specifier function &optional (priority 0))
  (set-pprint-dispatch type-specifier function priority *weave-pprint-dispatch*))
```

131. TEX-mode material is represented as a list of strings containing pure TEX text and lists of (inner-)Lisp forms, and this routine is responsible for printing it. It takes a *&rest* parameter so that it can be used with the '`~/.../`' *format* directive.

```
(defun print-tex (stream tex-mode-material &rest args)
  (declare (ignore args))
  (dolist (x tex-mode-material)
    (etypecase x
      (string (write-string x stream))
      (list (dolist (form x)
              (format stream "\\(~W\\)" form))))))
```

132. Control text (like section names) and comments are initially read as strings containing pure TEX text, but they actually contain restricted TEX-mode material, which may include inner-Lisp material. This routine re-reads such strings and picks up any inner-Lisp material.

```
(defun read-tex-from-string (input-string)
  (with-mode :restricted
    (with-input-from-string (stream input-string)
      (loop for text = (snarf-until-control-char stream #\|)
            for forms = (read-preserving-whitespace stream nil *eof* nil)
            if (plusp (length text)) collect text
            if (eof-p forms) do (loop-finish) else collect forms))))
```

133.

```
(defun print-limbo (stream section)
  (let ((commentary (section-commentary section)))
    (when commentary
      (print-tex stream commentary)
      (terpri stream))))

(set-weave-dispatch 'limbo-section #'print-limbo 1)
```

134.

```
(defun print-section (stream section)
  (format stream "~&\\~:[M~;N{1}~]{~:[~;T~]~D}" ; {1} should be depth
          (typep section 'starred-section)
          (typep section 'test-section)
          (section-number section))
  (let* ((commentary (section-commentary section))
         (name (section-name section))
         (named-section (and name (find-section name)))
         (code (section-code section)))
    (print-tex stream commentary)
    (fresh-line stream)
    (cond ((and commentary code) (format stream "\\Y\\B~%"))
          (code (format stream "\\B~%")))
    (when named-section
      (print-section-name stream named-section)
      (format stream "${}~:[\\mathrel+~;~]\\E{}$~%"
              (= (section-number section) (section-number named-section))))
    (when code
      (dolist (form code)
        (if (list-marker-p form)
          (format stream "~@<\\+~@;~W~;\\cr~:>" form)
          (format stream "~W~:[\\par~;~]" form (newlinep form))))
      (format stream "~&\\egroup~%")) ; matches \bgroup in \B
    (when (and (typep section 'test-section) (section-code section))
      (format stream "\\T~P~D.~%"
              (length (section-code section))
              (section-number (test-for-section section))))
    (when (and named-section
               (= (section-number section)
                  (section-number named-section)))
      (print-xrefs stream #\A
                   (remove section (named-section-sections named-section)))
      (print-xrefs stream #\U
                   (remove section (used-by named-section)))
      (print-xrefs stream #\Q
                   (remove section (cited-by named-section)))))
  (format stream "\\fi~%"))
(set-weave-dispatch 'section #'print-section)
```

135. The cross-references lists use the macros `\A` (for 'also'), `\U` (for 'use'), `\Q` (for 'quote'), and their pluralized variants, along with the conjunction macros `\ET` (for two section numbers) and `\ETs` (for between the last of three or more).

```
(defun print-xrefs (stream kind xrefs)
  (when xrefs
    ;; This was 16 lines of code over two sections in CWEB. I love format.
    (format stream "\\~C~{~#[~;~D~;s␣~D\\ET~D~:;s~@{~#[~;\\ETs~D~;~D~:;~D,␣~]~}~]~}.~%"
            kind (sort (mapcar #'section-number xrefs) #'<))))
```

136. Aside from printing the section name in the body of the woven output, this routine also knows how to print entries for the section name index, which uses a similar, but slightly different format.

```
(defun print-section-name (stream section &key (indexing nil))
  (format stream "~:[~;\\I~]\\X~{~D~^,␣~}:~/clweb::print-TeX/\\X"
          indexing
          (if indexing
              (sort (mapcar #'section-number (named-section-sections section)) #'<)
              (list (section-number section)))
          (read-tex-from-string (section-name section))))
(set-weave-dispatch 'named-section #'print-section-name)
```

137. When printing the section name index, the weaver wraps each named section in a *section-name-index* instance so that we can dispatch on that type.

```
(defclass section-name-index-entry ()
  ((named-section :accessor named-section :initarg :named-section)))
(set-weave-dispatch 'section-name-index-entry
  (λ (stream section-name &aux (section (named-section section-name)))
     (print-section-name stream section :indexing t)
     (terpri stream)
     (print-xrefs stream #\U (remove section (used-by section)))
     (print-xrefs stream #\Q (remove section (cited-by section)))))
```

138. Because we're outputting TeX, we need to carefully escape characters that TeX treats as special. Unfortunately, because TeX's syntax is so malleable (not unlike Lisp's), it's nontrivial to decide what to escape, how, and when.

The following routine is the basis for most of the escaping. It writes *string* to the output stream designated by *stream*, escaping the characters given in the a-list ∗*print-escape-list*∗. The entries in this a-list should be of the form '(⟨*characters*⟩ . ⟨*replacement*⟩)', where ⟨*replacement*⟩ describes how to escape each of the characters in ⟨*characters*⟩. Suppose c is a character in *string* that appears in one of the ⟨*characters*⟩ strings. If the corresponding ⟨*replacement*⟩ is a single character, then *print-escaped* will output it prior to every occurrence of c. If ⟨*replacement*⟩ is a string, it will be output *instead of* every occurrence of c in the input. Otherwise, c will be output without escaping.

The default value for ∗*print-escape-list*∗ defined below is suitable for escaping most TeX metacharacters; callers should bind it to a new value if they require specialized escaping.

```
(defparameter *print-escape-list*
  '(("␣\\%&#$^_~<>" . #\\) ("{" . "$\\{$") ("}" . "$\\}$")))
(defun print-escaped (stream string &rest args &aux
                      (stream (case stream
                                ((t) *terminal-io*)
                                ((nil) *standard-output*)
                                (otherwise stream))))
  (declare (ignore args))
  (loop for char across string
        as escape = (cdr (assoc char *print-escape-list* :test #'find))
        if escape
          do (etypecase escape
               (character (write-char escape stream)
                          (write-char char stream))
               (string (write-string escape stream)))
        else
          do (write-char char stream)))
```

139. We need to be careful about embedded newlines in string literals.

```
(defun print-string (stream string)
  (loop with *print-escape-list* = `(("{*}" . #\\)
                                     ("\\" . "\\\\\\\\")
                                     ("\"" . "\\\\\""")
                                     ,@*print-escape-list*)
        for last = 0 then (1+ newline)
        for newline = (position #\Newline string :start last)
        as line = (subseq string last newline)
        do (format stream "\\.{~:[~;\"~]~/clweb::print-escaped/~:[~;\"~]}"
                   (zerop last) line (null newline))
        when newline do (format stream "\\cr~:@_") else do (loop-finish)))
(set-weave-dispatch 'string #'print-string)
```

140.

```
(defun print-char (stream char)
  (let ((graphicp (and (graphic-char-p char) (standard-char-p char)))
        (name (char-name char))
        (*print-escape-list* `(("{}" . #\\) ,@*print-escape-list*)))
    (format stream "\\#\\CH{~/clweb::print-escaped/}"
            (if (and name (not graphicp))
                name
                (make-string 1 :initial-element char)))
    char))
(set-weave-dispatch 'character #'print-char)
```

141. Symbols are printed by first writing them out to a string, then printing that string. This relieves us of the burden of worrying about case conversion, package prefixes, and the like—we just let the Lisp printer handle it.

Lambda-list keywords and symbols in the 'keyword' package have specialized TEX macros that add a bit of emphasis.

```
(defun print-symbol (stream symbol)
  (let ((group-p (cond ((member symbol lambda-list-keywords)
                        (write-string "\\K{" stream))
                       ((keywordp symbol)
                        (write-string "\\:{" stream)))))
    (print-escaped stream (write-to-string symbol :escape nil :pretty nil))
    (when group-p (write-string "}" stream))))
(set-weave-dispatch 'symbol #'print-symbol)
```

142. Lambda gets special treatment.

```
(set-weave-dispatch '(eql λ)
  (λ (stream obj)
    (declare (ignore obj))
    (write-string "\\L" stream))
  1)
```

143. Next, we turn to list printing, and the tricky topic of indentation. On the assumption that the human writing a web is smarter than a program doing any sort of automatic indentation, we attempt to approximate (but not duplicate) on output the indentation given in the input by utilizing the character position values that the list reader stores in the list markers. We do this by breaking up lists into *logical blocks*—the same sort of (abstract) entities that the pretty printer uses, but made concrete here. A logical block defines a left edge for a list of forms, some of which may be nested logical blocks.

```
(defstruct (logical-block (:constructor make-logical-block (list)))
  list)
```

144. The analysis of the indentation is performed by a recursive *labels* routine, to which we will come in a moment. That routine operates on an list of *(form . charpos)* conses, taken from a list marker. Building this list does cost us some consing, but vastly simplifies the logic, since we don't have to worry about keeping multiple indices synchronized.

```
(defun analyze-indentation (list-marker)
  (declare (type list-marker list-marker))
  (labels ((find-next-newline (list) (member-if #'newlinep list :key #'car))
           (next-logical-block (list) ⟨ Build a logical block from list 145 ⟩))
    ;; Sanity check.
    (assert (= (length (list-marker-list list-marker))
               (length (list-marker-charpos list-marker)))
            ((list-marker-list list-marker) (list-marker-charpos list-marker))
            "List␣marker's␣list␣and␣charpos-list␣aren't␣the␣same␣length.")
    (next-logical-block (mapcar #'cons
                                (list-marker-list list-marker)
                                (list-marker-charpos list-marker)))))
```

145. To build a logical block, we identify groups of sub-forms that share a common left margin, which we'll call the *block indentation*. If that left margin is to the right of that of the current block, we recursively build a new subordinate logical block. The block ends when the first form on the next line falls to the left of the current block, or we run out of forms.

For example, consider the following simple form:

```
(first second
       third)
```

We start off with an initial logical block whose indentation is the character position of *first*: this is the *block indentation*. Then we look at *second*, and see that the first form on the following line, *third*, has the same position, and that it exceeds the current indentation level. And so we recurse, appending to the new logical block until we encounter a form whose indentation is less than the new block indentation, or, as in this trivial example, the end of the list.

More concretely, *next-logical-block* returns two values: the logical block constructed, and the unused portion of the list, which will always either be *nil* (we consumed the rest of the forms), or begin with a newline.

We keep a pointer to the next newline in *newline*, and *next-indent* is the indentation of the immediately following form, which will become the current indentation when we pass *newline*. As we do so, we store the difference between the current indentation and the block indentation in the newline marker, so that the printing routine, below, doesn't have to worry about character positions at all: it can just look at the newline markers and the logical block structure.

As an optimization, if we build a block that doesn't directly contain any newlines—a trivial logical block—we simply return a list of sub-forms instead of a logical block structure. This allows the printer to elide the alignment tabs, and makes the resulting TEX much simpler.

⟨ Build a logical block from *list* 145 ⟩ ≡

```
(do* ((block '())
      (block-indent (cdar list))
      (indent block-indent)
      (newline (find-next-newline list))
      (next-indent (cdadr newline)))
     ((or (endp list)
          (and (eq list newline) next-indent (< next-indent block-indent)))
      (values (if (notany #'newlinep block)
                  (nreverse block)
                  (make-logical-block (nreverse block)))
              list))
   (if (and indent next-indent
            (> next-indent indent)
            (= next-indent (cdar list)))
      (multiple-value-bind (sub-block tail) (next-logical-block list)
        (check-type (caar tail) (or newline-marker null))
        (push sub-block block)
        (setq list tail))
      (let ((next (car (pop list))))
        (push next block)
        (when (and list (newlinep next))
          (setf indent (cdar list)
                (indentation next) (- indent block-indent)
                newline (find-next-newline list)
                next-indent (cdadr newline))))))
```

This code is used in section 144.

146. Printing list markers is now simple, since the complexity is all in the logical blocks. . .

```
(defun print-list (stream list-marker)
  (let ((block (analyze-indentation list-marker)))
    (etypecase block
      (list (format stream "~<(~;~@{~W~^␣~}~;)~:>" block))
      (logical-block (format stream "(~W)" block)))))
(set-weave-dispatch 'list-marker #'print-list)
```

147. . . .but even here, it's not that bad. We start with a `\!`, which is just an alias for `\cleartabs`. Then we call (unsurprisingly) *pprint-logical-block*, using a per-line-prefix for our alignment tabs.

In the loop, we keep a one-token look-ahead to check for newlines, so that we can output separating spaces when and only when there isn't a newline coming up.

The logical-block building machinery above set the indentation on newlines to the difference in character positions of the first form following the newline and the block indentation. For differences of 1 or 2 columns, we use a single quad (`\1`); for any more, we use two (`\2`).

```
(defun print-logical-block (stream block)
  (write-string "\\!" stream)
  (pprint-logical-block (stream (logical-block-list block) :per-line-prefix "&")
    (do (indent
         next
         (obj (pprint-pop) next))
        (nil)
      (cond ((newlinep obj)
             (format stream "\\cr~:@_")
             (setq indent (indentation obj))
             (pprint-exit-if-list-exhausted)
             (setq next (pprint-pop)))
            (t (format stream "~@[~[~;\\1~;\\1~:;\\2~]~]~W" indent obj)
               (setq indent nil)
               (pprint-exit-if-list-exhausted)
               (setq next (pprint-pop))
               (unless (newlinep next)
                 (write-char #\␣ stream)))))))
(set-weave-dispatch 'logical-block #'print-logical-block)
```

148. Finally, we come to the printing of markers. These are all quite straightforward; the only thing to watch out for is the priorities. Because *pprint-dispatch* doesn't respect sub-type relationships, we need to set a higher priority for a sub-class than for its parent if we want a specialized pretty-printing routine.

Many of these routines output TEX macros defined in `clwebmac.tex`, which see.

149.

```
(set-weave-dispatch 'newline-marker
  (λ (stream obj)
    (declare (ignore obj))
    (terpri stream)))
(set-weave-dispatch 'par-marker
  (λ (stream obj)
    (declare (ignore obj))
    (format stream "~&\\Y~%"))
  1)
```

150.
```
(set-weave-dispatch 'empty-list-marker
  (λ (stream obj)
    (declare (ignore obj))
    (write-string "()" stream)))
(set-weave-dispatch 'consing-dot-marker
  (λ (stream obj)
    (declare (ignore obj))
    (write-char #\. stream)))
```

151.
```
(set-weave-dispatch 'quote-marker
  (λ (stream obj)
    (format stream "\\'~W" (quoted-form obj))))
```

152.
```
(set-weave-dispatch 'comment-marker
  (λ (stream obj)
    (format stream "\\C{~/clweb::print-TeX/}"
          (read-tex-from-string (comment-text obj)))))
```

153.
```
(set-weave-dispatch 'backquote-marker
  (λ (stream obj)
    (format stream "\\`~W" (backq-form obj))))
(set-weave-dispatch 'comma-marker
  (λ (stream obj)
    (format stream "\\CO{~@[~C~]}~W"
          (comma-modifier obj)
          (comma-form obj))))
```

154.
```
(set-weave-dispatch 'function-marker
  (λ (stream obj)
    (format stream "\\#\\'~S" (quoted-form obj)))
  1)
```

155.
```
(set-weave-dispatch 'simple-vector-marker
  (λ (stream obj)
    (format stream "\\#~@[~D~]~S"
            (and (slot-boundp obj 'length)
                 (slot-value obj 'length))
            (slot-value obj 'elements))))

(set-weave-dispatch 'bit-vector-marker
  (λ (stream obj)
    (format stream "\\#~@[~D~]*~{~[0~;1~]~}"
            (and (slot-boundp obj 'length)
                 (slot-value obj 'length))
            (map 'list #'identity
                  (slot-value obj 'elements))))
  1)
```

156.
```
(set-weave-dispatch 'read-time-eval-marker
  (λ (stream obj)
    (format stream "\\#.~W" (read-time-eval-form obj))))
```

157.
```
(set-weave-dispatch 'radix-marker
  (λ (stream obj)
    (format stream "$~VR_{~2:*~D}$"
            (radix-marker-base obj) (marker-value obj))))
```

158.
```
(set-weave-dispatch 'structure-marker
  (λ (stream obj)
    (format stream "\\#S~W" (structure-marker-form obj))))
```

159.
```
(set-weave-dispatch 'read-time-conditional-marker
  (λ (stream obj)
    (format stream "\\#~:[--~;+~]\\RC{~S␣~/clweb::print-escaped/}"
            (read-time-conditional-plusp obj)
            (read-time-conditional-test obj)
            (read-time-conditional-form obj))))
```

160. The walker. Our last major task is to produce an index of every interesting symbol that occurs in a web (we'll define what makes a symbol 'interesting' later). We would like separate entries for each kind of object that a given symbol names: e.g., local or global function, lexical or special variable, *&c.* And finally, we should note whether a given occurrence of a symbol is a definition/binding or just a use.

In order to accomplish this, we need to walk the tangled code of the entire program, since the meaning of a symbol in Common Lisp depends, in general, on its lexical environment, which can only be determined by code walking. For reasons that will be explained later, we'll actually be walking a special sort of munged code that isn't exactly Common Lisp. And because of this, none of the available code walkers will quite do what we want. So we'll roll our own.

161. We'll pass around instances of a *walker* class during our code walk to provide a hook (via subclassing) for overriding walker methods.

```
(defclass walker () ())
```

162. The walker protocol consists of a handful of generic functions, which we'll describe as we come to them.

⟨ Walker generic functions 168 ⟩

163. We'll use the environments API defined in CLtL-2, since even though it's not part of the Common Lisp standard, it's widely supported, and does everything we need it to do.

Allegro Common Lisp has an additional function, *ensure-portable-walking-environment*, that needs to be called on any environment object that a portable walker uses; we'll provide a trivial definition for other Lisps.

```
(unless (fboundp 'ensure-portable-walking-environment)
  (defun ensure-portable-walking-environment (env) env))
```

164. Allegro doesn't define *parse-macro* or *enclose*, so we'll need to provide our own definitions. In fact, we'll use our own version of *enclose* on all implementations (note that it's shadowed in the package definition at the beginning of the program). Thanks to Duane Rettig of Franz, Inc. for the idea behind this trivial implementation (post to comp.lang.lisp of 18 Oct, 2004, message-id ⟨*4is97u4vv.fsf@franz.com*⟩).

```
(defun enclose (lambda-expression &optional env (walker (make-instance 'walker)))
  (coerce (walk-form walker lambda-expression env) 'function))
```

165. The following code for *parse-macro* is obviously extremely implementation-specific; a portable version would be much more complex.

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (unless (fboundp 'parse-macro)
    (defun parse-macro (name lambda-list body &optional env)
      (declare (ignorable name lambda-list body env))
      #+:allegro (excl::defmacro-expander `(,name ,lambda-list ,@body) env)
      #-:allegro (error "PARSE-MACRO not implemented"))))
```

166. The good people at Franz also decided that they needed an extra value returned from both *variable-information* and *function-information*. But instead of doing the sensible thing and adding that extra value at the *end* of the list of returned values, they *changed* the order from the one specified by CLtL-2, so that their new value is the second value returned, and what should be second is now fourth. Thanks, Franz!

```
(defmacro reorder-env-information (fn orig)
  `(defun ,fn (&rest args)
     (multiple-value-bind (type locative declarations local) (apply ,orig args)
       (declare (ignore locative))
       (values type local declarations))))

#+:allegro (reorder-env-information variable-information #'sys:variable-information)
#+:allegro (reorder-env-information function-information #'sys:function-information)
```

167. The main entry point for the walker is *walk-form*. The walk ordinarily stops after encountering an atomic or special form; otherwise, we macro expand and try again. If a walker method wants to decline to walk, however, it may *throw* the given form, and the walk will continue with macro expansion.

```
(defmethod walk-form ((walker walker) form &optional env &aux
                      (env (ensure-portable-walking-environment env))
                      (expanded t))
  (flet ((symbol-macro-p (form env)
           (and (symbolp form)
                (eql (variable-information form env) :symbol-macro))))
    (loop
      (catch form
        (cond ((symbol-macro-p form env)) ; wait for macro expansion
              ((atom form)
               (return (walk-atomic-form walker form env)))
              ((not (symbolp (car form)))
               (return (walk-list walker form env)))
              ((or (not expanded)
                   (walk-as-special-form-p walker (car form) form env))
               (return (walk-compound-form walker (car form) form env)))))
      (multiple-value-setq (form expanded)
        (macroexpand-for-walk walker form env)))))
```

168. ⟨ Walker generic functions 168 ⟩ ≡

```
(defgeneric walk-form (walker form &optional env))
```

See also sections 170, 172, 174, 178, and 186.

This code is used in section 162.

169. The walker will treat a form as a special form if and only if *walk-as-special-form-p* returns true of that form.

```
(defmethod walk-as-special-form-p (walker operator form env)
  (declare (ignore walker operator form env))
  nil)
```

170. ⟨ Walker generic functions 168 ⟩ +≡

```
(defgeneric walk-as-special-form-p (walker operator form env))
```

171. Macroexpansion and environment augmentation both get wrapped in generic functions to allow subclasses to override the usual behavior. The default methods just call down to the normal Lisp functions.

```
(defmethod macroexpand-for-walk ((walker walker) form env)
  (macroexpand-1 form env))
(defmethod augment-walker-environment ((walker walker) env &rest args)
  (apply #'augment-environment env args))
```

172. ⟨ Walker generic functions 168 ⟩ +≡

```
(defgeneric macroexpand-for-walk (walker form env))
(defgeneric augment-walker-environment (walker env &rest args))
```

173. Here's a utility function we'll use often in walker methods: it walks each element of the supplied list and returns a new list of the results of those walks.

```
(defun walk-list (walker list env)
  (do ((form list (cdr form))
       (newform () (cons (walk-form walker (car form) env) newform)))
      ((atom form) (nreconc newform form))))
```

174. The functions *walk-atomic-form* and *walk-compound-form* are the real work-horses of the walker. The former takes a walker instance, an (atomic) form, an environment, and a flag indicating whether or not the form occurs in an evaluated position. Compound forms don't need such a flag, but we do provide an additional *operator* argument so that we can use *eql* specializers. The *operator* of a form passed to *walk-compound-form* will always be a symbol.

⟨ Walker generic functions 168 ⟩ +≡

```
(defgeneric walk-atomic-form (walker form env &optional evalp))
(defgeneric walk-compound-form (walker operator form env))
```

175. The default method for *walk-atomic-form* simply returns the form. Note that this function won't be called for symbol macros; those are expanded in *walk-form*.

```
(defmethod walk-atomic-form ((walker walker) form env &optional (evalp t))
  (declare (ignore env evalp))
  form)
```

176. The default method for *walk-compound-form* is used for all funcall-like forms; it leaves its *car* unevaluated and walks its *cdr*.

```
(defmethod walk-compound-form ((walker walker) operator form env)
  (declare (ignore operator))
  `(,(walk-atomic-form walker (car form) env nil)
    ,@(walk-list walker (cdr form) env)))
```

177. Common Lisp defines a *function name* as "[a] symbol or a list *(setf symbol)*." Since they're not necessarily atomic, we define a special walker function just for function names.

```
(deftype setf-function-name () '(cons (eql setf) (cons symbol null)))
(defmethod walk-function-name ((walker walker) function-name env &key)
  (typecase function-name
    (symbol (walk-atomic-form walker function-name env nil))
    (setf-function-name `(setf ,(walk-atomic-form walker (cadr function-name) env nil)))
    (t (cerror "Use␣the␣function␣name␣anyway." 'invalid-function-name :name function-name)
      function-name)))
```

178. ⟨ Walker generic functions 168 ⟩ +≡

```
(defgeneric walk-function-name (walker function-name env &key &allow-other-keys))
```

179. ⟨ Condition classes 27 ⟩ +≡

```
(define-condition invalid-function-name (parse-error)
  ((name :initarg :name :reader invalid-function-name))
  (:report (λ (error stream)
             (format stream "~@<Invalid␣function␣name␣~A.~:@>"
                     (invalid-function-name error)))))
```

180. Many of the special forms defined in Common Lisp can be walked using the default method for *walk-compound-form* just defined, since their syntax is the same as an ordinary function call. But it's important to override *walk-as-special-form-p* for these operators, because "[a]n implementation is free to implement a Common Lisp special operator as a macro." (ANSI Common Lisp, section 3.1.2.1.2.2)

```
(macrolet ((walk-as-special-form (operator)
             `(defmethod walk-as-special-form-p ((walker walker) (operator (eql ',operator)) form env)
                (declare (ignore form env))
                t)))
  (walk-as-special-form catch)
  (walk-as-special-form if)
  (walk-as-special-form load-time-value)
  (walk-as-special-form multiple-value-call)
  (walk-as-special-form multiple-value-prog1)
  (walk-as-special-form progn)
  (walk-as-special-form progv)
  (walk-as-special-form setq)
  (walk-as-special-form tagbody)
  (walk-as-special-form throw)
  (walk-as-special-form unwind-protect))
```

181. The rest of the special form walkers we define will need specialized methods for both *walk-as-special-form-p* and *walk-compound-form*. The following macro makes sure that these are consistently defined.

```
(defmacro define-special-form-walker (operator (walker form env &rest rest) &body body &aux
                                      (oparg `(,(gensym) (eql ',operator))))
  (flet ((arg-name (arg) (if (consp arg) (car arg) arg)))
    `(progn
       (defmethod walk-as-special-form-p (,walker ,oparg ,form ,env)
         (declare (ignorable ,@(mapcar #'arg-name `(,walker ,form ,env))))
         t)
       (defmethod walk-compound-form (,walker ,oparg ,form ,env ,@rest)
         (declare (ignorable ,@(mapcar #'arg-name `(,walker ,form ,env))))
         ,@body))))
```

182. Block-like special forms have a *cdr* that begins with a single unevaluated form, followed by zero or more evaluated forms.

```
(macrolet ((define-block-like-walker (operator)
             `(define-special-form-walker ,operator ((walker walker) form env)
                `(,(car form)
                  ,(walk-atomic-form walker (cadr form) env nil)
                  ,@(walk-list walker (cddr form) env)))))
  (define-block-like-walker block)
  (define-block-like-walker eval-when)
  (define-block-like-walker return-from)
  (define-block-like-walker the))
```

183. Quote-like special forms are entirely unevaluated.

```
(macrolet ((define-quote-like-walker (operator)
             `(define-special-form-walker ,operator ((walker walker) form env)
                (declare (ignore walker env))
                form)))
  (define-quote-like-walker quote)
  (define-quote-like-walker go))
```

184. The *function* special form takes either a valid function name or a λ expression.

```
(define-special-form-walker function ((walker walker) form env)
  `(,(car form)
    ,(if (and (consp (cadr form))
              (eql (caadr form) 'λ))
         (walk-lambda-expression walker (cadr form) env)
         (walk-function-name walker (cadr form) env))))
```

185. Next, we'll work our way up to parsing λ expressions and other function-defining forms.

Given a sequence of declarations and possibly a documentation string followed by other forms (as occurs in the bodies of *defun*, *defmacro*, *&c.*), *parse-body* returns *(values forms decls doc)*, where *decls* is the declaration specifiers found in each *declare* expression, *doc* holds a doc string (or *nil* if there is none), and *forms* holds the other forms. See ANSI Common Lisp section 3.4.11 for the rules on the syntactic interaction of doc strings and declarations.

If *doc-string-allowed* is *nil* (the default), then no forms will be treated as documentation strings.

```
(defun parse-body (body &key doc-string-allowed walker env &aux doc)
  (flet ((doc-string-p (x remaining-forms)
           (and (stringp x) doc-string-allowed remaining-forms (null doc)))
         (declaration-p (x)
           (and (listp x) (eql (car x) 'declare))))
    (loop for forms = body then (cdr forms)
          as x = (car forms)
          while forms
          if (doc-string-p x (cdr forms)) do (setq doc x)
          else if (declaration-p x) append (cdr x) into decls
               else do (loop-finish)
          finally (return (values forms
                                  (if walker
                                      (walk-declaration-specifiers walker decls env)
                                      decls)
                                  doc)))))
```

186. Declaration forms aren't evaluated in Common Lisp, but we'll want to be able to walk the specifiers.

```
⟨ Walker generic functions 168 ⟩ +≡
(defgeneric walk-declaration-specifiers (walker decls env))
```

187.

```
(defmethod walk-declaration-specifiers ((walker walker) decls env)
  (declare (ignore env))
  decls)
```

188. The syntax of λ-lists is given in section 3.4 of the ANSI standard. We accept the syntax of macro λ-lists, since they are the most general, and appear to be a superset of every other type of λ-list.

This function returns two values: the walked λ-list and a new environment object containing bindings for all of the parameters found therein.

```
(defun walk-lambda-list (walker lambda-list decls env &aux
                         new-lambda-list (state :reqvars))
  (labels ((augment-env (&rest vars &aux (vars (remove-if #'null vars)))
             (setq env (augment-walker-environment walker env :variable vars :declare decls)))
           (walk-var (var)
             (walk-atomic-form walker var env nil))
           (update-state (keyword)
             (setq state (ecase keyword
                           ((nil) state)
                           (&optional :optvars)
                           ((&rest &body) :restvar)
                           (&key :keyvars)
                           (&aux :auxvars)
                           (&environment :envvar))))
           (maybe-destructure (var/pattern)
             (if (consp var/pattern)
                 (walk-lambda-list walker var/pattern decls env)
                 (values (walk-var var/pattern)
                         (augment-env var/pattern)))))
    ⟨ Check for &whole and &environment vars, and augment the lexical environment with them if found 189 ⟩
    (do* ((lambda-list lambda-list (cdr lambda-list))
          (arg (car lambda-list) (car lambda-list)))
         ((null lambda-list) (values (nreverse new-lambda-list) env))
      (ecase state
        (:envvar ⟨ Process arg as an environment parameter 190 ⟩)
        ((:reqvars :restvar) ; required and rest parameters have the same syntax
         ⟨ Process arg as a required parameter 191 ⟩)
        (:optvars ⟨ Process arg as an optional parameter 192 ⟩)
        (:keyvars ⟨ Process arg as a keyword parameter 193 ⟩)
        (:auxvars ⟨ Process arg as an auxiliary variable 194 ⟩))
      (when (or (atom lambda-list)
                (and (cdr lambda-list)
                     (atom (cdr lambda-list))))
        ⟨ Process dotted rest var 196 ⟩))))
```

189. A *&whole* variable must come first in a λ-list, and an *&environment* variable, although it may appear anywhere in the list, must be bound along with *&whole*. We'll pop a *&whole* variable off the front of *lambda-list*, but we'll leave any *&environment* variable to be picked up later.

⟨ Check for *&whole* and *&environment* vars, and augment the lexical environment with them if found 189 ⟩ ≡

```
(let ((wholevar (and (consp lambda-list)
                     (eql (car lambda-list) '&whole)
                     (push (pop lambda-list) new-lambda-list)
                     (car (push (walk-var (pop lambda-list)) new-lambda-list))))
      (envvar (do ((lambda-list lambda-list (cdr lambda-list)))
                  ((atom lambda-list) nil)
                (when (eql (car lambda-list) '&environment)
                  (return (cadr lambda-list))))))
  (augment-env wholevar envvar))
```

This code is used in section 188.

190. We've already added the environment variable to our lexical environment, so we just push it onto the new λ-list and prepare for the next parameter.

⟨ Process *arg* as an environment parameter 190 ⟩ ≡

```
(push (walk-var (pop lambda-list)) new-lambda-list)
(when (consp lambda-list)
  (update-state (car lambda-list)))
```

This code is used in section 188.

191. ⟨ Process *arg* as a required parameter 191 ⟩ ≡

```
(etypecase arg
  (symbol ⟨ Process the symbol in arg as a parameter 195 ⟩)
  (cons
   (multiple-value-bind (pattern new-env)
         (walk-lambda-list walker arg decls env)
     (setq env new-env)
     (push pattern new-lambda-list))))
```

This code is used in section 188.

192. Watch the order here: in the non-simple-*var* case, both the init form and the pattern (if any) need to be walked in an environment *unaugmented* with any supplied-p-parameter.

⟨ Process *arg* as an optional parameter 192 ⟩ ≡

```
(etypecase arg
  (symbol ⟨ Process the symbol in arg as a parameter 195 ⟩)
  (cons
   (destructuring-bind (var/pattern &optional init-form supplied-p-parameter) arg
     (when init-form
       (setq init-form (walk-form walker init-form env)))
     (multiple-value-setq (var/pattern env)
       (maybe-destructure var/pattern))
     (when supplied-p-parameter
       (augment-env supplied-p-parameter))
     (push (nconc (list var/pattern)
                  (and init-form (list init-form))
                  (and supplied-p-parameter (list supplied-p-parameter)))
           new-lambda-list))))
```

This code is used in section 188.

193. ⟨Process *arg* as a keyword parameter 193⟩ ≡

```
(cond ((eql arg '&allow-other-keys)
       (push (pop lambda-list) new-lambda-list)
       (update-state (car lambda-list)))
      (t
       (etypecase arg
         (symbol ⟨Process the symbol in arg as a parameter 195⟩)
         (cons
          (destructuring-bind (var/kv &optional init-form supplied-p-parameter) arg
            (when init-form
              (setq init-form (walk-form walker init-form env)))
            (cond ((consp var/kv)
                   (destructuring-bind (keyword-name var/pattern) var/kv
                     (multiple-value-setq (var/pattern env)
                       (maybe-destructure var/pattern))
                     (setq var/kv (list (walk-var keyword-name) var/pattern))))
                  (t (setq var/kv (walk-var var/kv))
                     (augment-env var/kv)))
            (when supplied-p-parameter
              (setq supplied-p-parameter (walk-var supplied-p-parameter))
              (augment-env supplied-p-parameter))
            (push (nconc (list var/kv)
                         (and init-form (list init-form))
                         (and supplied-p-parameter (list supplied-p-parameter)))
                  new-lambda-list))))))
```

This code is used in section 188.

194. ⟨Process *arg* as an auxiliary variable 194⟩ ≡

```
(etypecase arg
  (symbol ⟨Process the symbol in arg as a parameter 195⟩)
  (cons
   (destructuring-bind (var &optional init-form) arg
     (setq var (walk-var var)
           init-form (and init-form (walk-form walker init-form env)))
     (augment-env var)
     (push (nconc (list var)
                  (and init-form (list init-form)))
           new-lambda-list))))
```

This code is used in section 188.

195. ⟨Process the symbol in *arg* as a parameter 195⟩ ≡

```
(cond ((member arg lambda-list-keywords)
       (push arg new-lambda-list)
       (update-state arg))
      (t (setq arg (walk-var arg))
         (augment-env arg)
         (push arg new-lambda-list)))
```

This code is used in sections 191, 192, 193, and 194.

196. We normalize a dotted rest parameter into a proper *&rest* parameter to avoid having to worry about reversing an improper *new-lambda-list*.

⟨ Process dotted rest var 196 ⟩ ≡
```
(let ((var (walk-var (if (consp lambda-list) (cdr lambda-list) lambda-list))))
   (augment-env var)
   (push '&rest new-lambda-list)
   (push var new-lambda-list))
(setq lambda-list nil)
```
This code is used in section 188.

197. While we're in the mood to deal with λ-lists, here's a routine that can walk the 'specialized' λ-lists used in *defmethod* forms. We can't use the same function as the one used to parse macro λ-lists, since the syntax would be ambiguous: there would be no way to distinguish between a specialized required parameter and a destructuring pattern. What we can do, however, is peel off just the required parameters from a specialized λ-list, and feed the rest to *walk-lambda-list*.

This function returns the same kind of values as *walk-lambda-list*; viz., the walked specialized λ-list and an environment augmented with the parameters found therein.

```
(deftype class-specializer () '(cons symbol (cons symbol null)))
(deftype compound-specializer (&optional (operator 'eql))
   `(cons symbol (cons (cons (eql ,operator) *) null)))
(defun walk-specialized-lambda-list (walker lambda-list decls env)
   (let ((req-params ⟨ Extract the required parameters from lambda-list 198 ⟩))
      (multiple-value-bind (other-params env) (walk-lambda-list walker lambda-list decls env)
         (values (nconc req-params other-params) env))))
```

198. ⟨ Extract the required parameters from *lambda-list* 198 ⟩ ≡
```
(flet ((augment-env (var)
            (setq env (augment-walker-environment walker env :variable (list var) :declare decls)))
        (walk-var (spec)
            (etypecase spec
               (symbol (walk-atomic-form walker spec env nil))
               (class-specializer
                (list (walk-atomic-form walker (car spec) nil)
                      (walk-atomic-form walker (cadr spec) nil)))
               ((compound-specializer eql)
                (list (walk-atomic-form walker (car spec) nil)
                      `(eql ,(walk-form walker (cadadr spec))))))))
   (loop until (or (null lambda-list)
                   (member (car lambda-list) lambda-list-keywords))
         as var = (walk-var (pop lambda-list))
         do (augment-env (if (consp var) (car var) var))
         collect var))
```
This code is used in section 197.

199. Having built up the necessary machinery, walking a λ expression is now straightforward. The slight generality of walking the car of the form using *walk-function-name* is because this function will also be used to walk the bindings in *flet*, *macrolet*, and *labels* special forms. Also note that this function passes all of its extra arguments down to *walk-function-name*; this will be used later to aid in the indexing process.

```
(defun walk-lambda-expression (walker form env &rest args &aux
                                         (lambda-list (cadr form)) (body (cddr form)))
  (multiple-value-bind (forms decls doc) (parse-body body :walker walker :env env :doc-string-allowed t)
    (multiple-value-bind (lambda-list env) (walk-lambda-list walker lambda-list decls env)
      `(,(apply #'walk-function-name walker (car form) env args)
        ,lambda-list
        ,@(if doc `(,doc))
        ,@(if decls `((declare ,@decls)))
        ,@(walk-list walker forms env)))))
```

200. 'lambda' is not a special operator in Common Lisp, but we'll treat λ expressions as special forms.

```
(define-special-form-walker λ ((walker walker) form env)
  (walk-lambda-expression walker form env))
```

201. We come now to the binding special forms. The six binding special forms in Common Lisp (*let*, *let∗*, *flet*, *labels*, *macrolet*, and *symbol-macrolet*) all have essentially the same syntax; only the scope and namespace of the bindings differ.

We'll start with a little utility routine that walks a variable-like binding form (e.g., the *cadr* of a *let*/*let∗* or *symbol-macrolet* form). It normalizes atomic binding forms to conses in order to avoid special cases in the actual walker methods.

The function-like binding forms will use *walk-lambda-expression* for the same purpose.

```
(defun walk-variable-binding (walker p env &aux (binding (if (consp p) p (list p))))
  (list (walk-atomic-form walker (car binding) env nil)
        (and (cdr binding)
             (walk-form walker (cadr binding) env))))
```

202. *let*, *flet*, *macrolet*, and *symbol-macrolet* are all 'parallel' binding forms: they walk their bindings in an unaugmented environment, then execute their body forms in an environment that contains all of the new bindings.

```
(define-special-form-walker let
      ((walker walker) form env &aux
       (bindings (mapcar (λ (p) (walk-variable-binding walker p env)) (cadr form)))
       (body (cddr form)))
  (multiple-value-bind (forms decls) (parse-body body :walker walker :env env)
    `(,(car form)
      ,bindings
      ,@(if decls `((declare ,@decls)))
      ,@(walk-list walker forms
                   (augment-walker-environment walker env
                                               :variable (mapcar #'car bindings)
                                               :declare decls)))))
```

203.

```
(define-special-form-walker flet
      ((walker walker) form env &aux
       (bindings (mapcar (λ (p) (walk-lambda-expression walker p env :operator 'flet :local t :def t))
                         (cadr form)))
       (body (cddr form)))
   (multiple-value-bind (forms decls) (parse-body body :walker walker :env env)
     `(,(car form)
       ,bindings
       ,@(if decls `((declare ,@decls)))
       ,@(walk-list walker forms
                    (augment-walker-environment walker env
                                                :function (mapcar #'car bindings)
                                                :declare decls)))))
```

204. The bindings established by *macrolet* and *symbol-macrolet* are different from those established by the other binding forms in that they include definitions as well as names. We'll use a little helper function, *make-macro-definitions*, for building the expander functions in the former case.

```
(defun make-macro-definitions (walker defs env)
   (mapcar (λ (def &aux (name (walk-atomic-form walker (car def) env nil)))
             (list name (enclose (parse-macro name (cadr def) (cddr def) env) env walker)))
           defs))

(define-special-form-walker macrolet
      ((walker walker) form env &aux
       (bindings (mapcar (λ (p) (walk-lambda-expression walker p env :operator 'macrolet :local t :def t))
                         (cadr form)))
       (body (cddr form)))
   (multiple-value-bind (forms decls) (parse-body body :walker walker :env env)
     `(,(car form)
       ,bindings
       ,@(if decls `((declare ,@decls)))
       ,@(walk-list walker forms
                    (augment-walker-environment walker env
                                                :macro (make-macro-definitions walker bindings env)
                                                :declare decls)))))
```

205. Walking *symbol-macrolet* is simpler, since the definitions are given in the bindings themselves.

```
(define-special-form-walker symbol-macrolet
      ((walker walker) form env &aux
       (bindings (mapcar (λ (p) (walk-variable-binding walker p env)) (cadr form)))
       (body (cddr form)))
   (multiple-value-bind (forms decls) (parse-body body :walker walker :env env)
     `(,(car form)
       ,bindings
       ,@(if decls `((declare ,@decls)))
       ,@(walk-list walker forms
                    (augment-walker-environment walker env
                                                :symbol-macro bindings
                                                :declare decls)))))
```

206. The two outliers are *let∗*, which augments its environment sequentially, and *labels*, which does so before walking any of its bindings.

```
(define-special-form-walker let∗
      ((walker walker) form env &aux
       (bindings (cadr form))
       (body (cddr form)))
   (multiple-value-bind (forms decls) (parse-body body :walker walker :env env)
      ‘(,(car form)
        ,(mapcar (λ (p &aux
                        (walked-binding (walk-variable-binding walker p env))
                        (variable (list (car walked-binding))))
                       (setq env (augment-walker-environment walker env
                                                              :variable variable
                                                              :declare decls))
                       walked-binding)
                   bindings)
        ,@(if decls ‘((declare ,@decls)))
        ,@(walk-list walker forms env))))
```

207.

```
(define-special-form-walker labels
      ((walker walker) form env &aux
       (bindings (cadr form))
       (body (cddr form)))
   (multiple-value-bind (forms decls) (parse-body body :walker walker :env env)
      (let∗ ((function-names (mapcar (λ (p)
                                          (walk-function-name walker (car p) env
                                                                :operator 'labels
                                                                :local t
                                                                :def t))
                                        bindings))
             (env (augment-walker-environment walker env
                                                :function function-names
                                                :declare decls)))
        ‘(,(car form)
          ,(mapcar (λ (p) (walk-lambda-expression walker p env)) bindings)
          ,@(if decls ‘((declare ,@decls)))
          ,@(walk-list walker forms env)))))
```

208. The last special form we need a specialized walker for is *locally*, which simply executes its body in a lexical environment augmented by a set of declarations.

```
(define-special-form-walker locally ((walker walker) form env)
   (multiple-value-bind (forms decls) (parse-body (cdr form) :walker walker :env env)
      ‘(,(car form)
        ,@(if decls ‘((declare ,@decls)))
        ,@(walk-list walker forms
                      (augment-walker-environment walker env :declare decls)))))
```

209. Indexing. Having constructed our code walker, we can now use it to produce an index of the symbols in a web. We'll say a symbol is *interesting* if it is interned in one of the packages listed in **index-packages**. The user should add packages to this list using the `@x` control code, which calls the following function. It takes a designator for a list of package designators, and adds each package to **index-packages**.

```
(defun index-package (packages &aux (packages (ensure-list packages)))
  "Inform␣the␣weaver␣that␣it␣should␣index␣the␣symbols␣in␣PACKAGES."
  (dolist (package packages)
    (pushnew (find-package package) *index-packages*)))
```

210. ⟨ Global variables 12 ⟩ +≡

```
(defvar *index-packages* nil)
```

211. ⟨ Initialize global variables 13 ⟩ +≡

```
(setq *index-packages* nil)
```

212. Before we proceed, let's establish some terminology. Formally, an *index* is an ordered collection of *entries*, each of which is a (⟨*heading*⟩, ⟨*locator*⟩) pair: the *locator* indicates where the object referred to by the *heading* may be found. A list of entries with the same heading is called an *entry list*, or sometimes just an *entry*; the latter is an abuse of terminology, but useful and usually clear in context.

Headings may in general be multi-leveled, and are sorted lexicographically. In this program, headings are represented by (designators for) lists of objects that have a specialized *heading-name* method, which method should return a string designator.

```
(defun entry-heading-lessp (h1 h2 &aux (h1 (ensure-list h1)) (h2 (ensure-list h2)))
  (or (and (null h1) h2)
      (string-lessp (heading-name (car h1)) (heading-name (car h2)))
      (and (string-equal (heading-name (car h1)) (heading-name (car h2)))
           (cdr h2)
           (entry-heading-lessp (cdr h1) (cdr h2)))))
(defun entry-heading-equalp (h1 h2 &aux (h1 (ensure-list h1)) (h2 (ensure-list h2)))
  (and (= (length h1) (length h2))
       (every #'string-equal
              (mapcar #'heading-name h1)
              (mapcar #'heading-name h2))))
```

213. Generally, those objects will be instances of the class *heading* or one of its sub-classes, but we'll allow a few other types of objects, too. The two sub-classes defined here are, respectively, for headings that should be printed in `typewriter type`, and ones that should be printed under the control of the TEX macro '`\9`', which the user can define as desired.

```
(defclass heading ()
  ((name :initarg :name)))
(defclass tt-heading (heading) ())
(defclass custom-heading (heading) ())
```

214. To simplify some of the logic concerning heading names, we'll use a custom method combination for *heading-name*.

```
(defun concatenate-string (&rest args) (apply #'concatenate 'string args))
(define-method-combination concatenate-string :identity-with-one-argument t)
```

215. The default *heading-name* method for *heading* instances is the obvious one. Symbols and strings are valid headings, too.

```
(defgeneric heading-name (heading)
  (:method-combination concatenate-string)
  (:method concatenate-string ((heading heading)) (slot-value heading 'name))
  (:method concatenate-string ((heading symbol)) heading)
  (:method concatenate-string ((heading string)) heading))
```

216. The following heading classes are all used as sub-headings, and represent the namespace in which the object referred to by the primary heading is located.

```
(defclass global/local-heading (heading)
  ((local :initarg :local))
  (:default-initargs :local nil))
(defclass function-heading (global/local-heading)
  ((generic :initarg :generic))
  (:default-initargs :name "function" :generic nil))
(defclass setf-function-heading (function-heading) ()
  (:default-initargs :name "setf␣function"))
(defclass macro-heading (global/local-heading) ()
  (:default-initargs :name "macro"))
(defclass symbol-macro-heading (global/local-heading) ()
  (:default-initargs :name "symbol␣macro"))
(defclass method-heading (heading)
  ((qualifiers :reader method-heading-qualifiers :initarg :qualifiers))
  (:default-initargs :name "method" :qualifiers nil))
(defclass setf-method-heading (method-heading) ()
  (:default-initargs :name "setf␣method"))
(defclass variable-heading (heading)
  ((special :initarg :special)
   (constant :initarg :constant))
  (:default-initargs :name "variable" :special nil :constant nil))
(defclass class-heading (heading) ()
  (:default-initargs :name "class"))
(defclass condition-class-heading (heading) ()
  (:default-initargs :name "condition␣class"))
```

217. Several of the heading classes have slots whose names should be prepended to the corresponding heading names if the value stored there is true; e.g., if a function sub-heading is marked as local, then its name should be "local function".

```
(defmacro define-heading-name-prefix (class &rest slot-names)
  (let ((*print-case* :downcase)
        (*print-escape* nil)
        (*print-pretty* nil))
    `(defmethod heading-name concatenate-string ((heading ,class))
       (with-slots ,slot-names heading
         (cond ,@(loop for slot-name in slot-names
                       collect `(,slot-name ,(format nil "~A␣" slot-name))))))))
(define-heading-name-prefix global/local-heading local)
(define-heading-name-prefix function-heading generic)
(define-heading-name-prefix variable-heading constant special)
```

218. For method headings, we'll prepend the list of qualifiers if present; otherwise we'll call them 'primary'.

```
(defmethod heading-name concatenate-string ((heading method-heading))
  (format nil "~:[primary~;~:*~{~A~^␣~}~]␣" (method-heading-qualifiers heading)))
```

219. The sub-heading classes above are usually constructed by the following function during indexing. The keyword arguments are passed down from the various walker functions, and this function passes them down to the initializers for the various (sub-)heading classes, but peeks at a few of them to figure out what kind of heading class to instantiate.

```
(defun make-sub-heading (operator &rest args &key function-name qualifiers &allow-other-keys)
  (apply #'make-instance
         (ecase operator
           ((nil defgeneric) 'function-heading)
           ((defmethod) ⟨ Choose an appropriate heading class for a method 220 ⟩)
           ((defun flet labels) (typecase function-name
                                  (symbol 'function-heading)
                                  (setf-function-name 'setf-function-heading)))
           ((defmacro macrolet) 'macro-heading)
           ((defvar defparameter defconstant) 'variable-heading)
           ((defclass) 'class-heading)
           ((define-condition) 'condition-class-heading))
         :allow-other-keys t
         args))
```

220. Non-*setf* primary methods (i.e., methods with empty qualifier lists) are indexed as functions so that they'll appear as definitions of the generic function they're methods of; only methods with qualifiers and *setf* methods get independent entries. This helps keep the index compact without significantly affecting usability.

```
⟨ Choose an appropriate heading class for a method 220 ⟩ ≡
(typecase function-name
  (symbol (if qualifiers 'method-heading 'function-heading))
  (setf-function-name 'setf-method-heading))
```

This code is used in section 219.

221. Now let's turn our attention to the other half of index entries. In this program, a locator is either a pointer to a section (the usual case) or a cross-reference to another index entry. We'll represent locators as instances of a *locator* class, and use a single generic function, *location*, to dereference them.

Section locators have an additional slot for a definition flag, which when true indicates that the object referred to by the associated heading is defined in the section represented by that locator, not just used. Such locators will be given a bit of typographic emphasis by the weaver when it prints the containing entry.

```
(defclass locator () ())
(defclass section-locator (locator)
  ((section :accessor location :initarg :section)
   (def :accessor locator-definition-p :initarg :def)))
(defclass xref-locator (locator)
  ((heading :accessor location :initarg :heading)))
(defclass see-locator (xref-locator) ())
(defclass see-also-locator (xref-locator) ())
```

222. Here's a constructor for the various kinds of locators.

```
(defun make-locator (&key section def see see-also)
  (assert (if (or see see-also) (and (not section) (not def)) t)
          (section def see see-also)
          "Can't␣use␣:SECTION␣or␣:DEF␣with␣:SEE␣or␣:SEE-ALSO.")
  (assert (if def section t) (section def) "Can't␣use␣:DEF␣without␣:SECTION.")
  (assert (not (and see see-also)) (see see-also) "Can't␣use␣both␣:SEE␣and␣:SEE-ALSO.")
  (cond (section (make-instance 'section-locator :section section :def def))
        (see (make-instance 'see-locator :heading see))
        (see-also (make-instance 'see-locator :heading see-also))))
```

223. Since we'll eventually want the index sorted by heading, we'll store the entries in a binary search tree. To simplify processing, what we'll actually store is *entry lists*, which are collections of entries with identical headings, but we'll overload the term in what seems to be a fairly traditional manner and call them entries, too.

```
(defclass index-entry (binary-search-tree)
  ((key :accessor entry-heading)
   (locators :accessor entry-locators :initarg :locators :initform '())))
(defmethod find-or-insert (item (root index-entry) &key
                               (predicate #'entry-heading-lessp)
                               (test #'entry-heading-equalp)
                               (insert-if-not-found t))
  (call-next-method item root
                    :predicate predicate
                    :test test
                    :insert-if-not-found insert-if-not-found))
```

224. The entry trees get stored in *index* objects, for which we define a few protocol functions: *make-index* creates a new index; *add-index-entry* adds an entry to it, and *find-index-entries* returns the list of locators with the given heading.

```
(defclass index ()
  ((entries :accessor index-entries :initform nil)))
(defun make-index () (make-instance 'index))
(defgeneric add-index-entry (index heading locator &key))
(defgeneric find-index-entries (index heading))
```

225. We'll keep a global index around so that we can add 'manual' entries (i.e., entries not automatically generated via the code walk) during reading.

```
⟨ Global variables 12 ⟩ +≡
(defvar *index* nil)
```

226. ⟨ Initialize global variables 13 ⟩ +≡

```
(setq *index* (make-index))
```

227. This method adds an index entry for *heading* with location *section*. A new locator is constructed only when necessary, and duplicate locators are automatically suppressed. Definitional locators are also made to supersede ordinary ones.

```
(define-modify-macro orf (&rest args) or)
(defmethod add-index-entry ((index index) heading (section section) &key def &aux
                               (heading (ensure-list heading)))
   (flet ((make-locator ()
            (make-locator :section section :def def)))
      (if (null (index-entries index))
         (setf (index-entries index)
               (make-instance 'index-entry :key heading :locators (list (make-locator))))
         (let* ((entry (find-or-insert heading (index-entries index)))
                (old-locator (find section (entry-locators entry) :key #'location)))
            (if old-locator
                (orf (locator-definition-p old-locator) def)
                (push (make-locator) (entry-locators entry)))))))
```

228. And here's the main index entry retrieval method. In fact, this function isn't actually used in this program (although the test suite uses it), since all we do with the index is add entries to it and then traverse the whole thing and print them all out.

```
(defmethod find-index-entries ((index index) heading)
   (let ((entries (index-entries index)))
      (when entries
         (multiple-value-bind (entry present-p) (find-or-insert heading entries :insert-if-not-found nil)
            (when present-p
               (entry-locators entry))))))
```

229. In order to index them properly, the walker needs to know whether a given function name refers to an ordinary or generic function.

```
(defun generic-function-p (function-name)
   (or (typep function-name 'generic-function)
       (and (fboundp function-name)
            (typep (fdefinition function-name) 'generic-function))))
```

230. Here are a couple of indexing routines that we'll use in the walker below for indexing function and variable *uses*. They look up the given variable or function in a lexical environment object and add an appropriate index entry. (We can't use this routine for definitions because the name being indexed might not be in the environment yet.) We don't currently index lexical variables, as that would probably just bulk up the index to no particular advantage.

```
(defun index-variable (index variable section env)
  (multiple-value-bind (type local) (variable-information variable env)
    (when (member type '(:special :symbol-macro :constant))
      (add-index-entry
       index
       (list variable
             (ecase type
               (:special (make-instance 'variable-heading :special t))
               (:constant (make-instance 'variable-heading :constant t))
               (:symbol-macro (make-instance 'symbol-macro-heading :local local))))
       section))))
(defun index-funcall (index function-name section env)
  (multiple-value-bind (type local) (function-information function-name env)
    (when (member type '(:function :macro))
      (add-index-entry
       index
       (list function-name
             (ecase type
               (:function (make-instance 'function-heading
                                         :local local
                                         :generic (generic-function-p function-name)))
               (:macro (make-instance 'macro-heading :local local))))
       section))))
```

231. To index definitions, we'll need more information from the walker about the context in which the variable or function name occurred. In particular, we'll get the *car* of the defining form as the *operator* argument, which is then passed down to *make-sub-heading*.

```
(defun index-defvar (index variable section &key operator special)
  (add-index-entry
   index
   (list variable (make-sub-heading operator
                                    :special special
                                    :constant (eql operator 'defconstant)))
   section
   :def t))

(defun index-defun (index function-name section &key operator local generic qualifiers)
  (add-index-entry
   index
   (list (typecase function-name
           (symbol function-name)
           (setf-function-name (cadr function-name)))
         (make-sub-heading operator
                           :function-name function-name
                           :local local
                           :generic (or generic (generic-function-p function-name))
                           :qualifiers qualifiers))
   section
   :def t))
```

232. We'll perform the indexing by walking over the code of each section and noting each of the interesting symbols that we find there according to its semantic role. In theory, this should be a straightforward task for any Common Lisp code walker. What makes it tricky is that references to named sections can occur anywhere in a form, which might break the syntax of macros and special forms unless we tangle the form first. But once we tangle a form, we lose the provenance of the sub-forms that came from named sections, and so our index would be wrong.

The trick that we use to overcome this problem is to tangle the forms in a special way where instead of just splicing the named section code into place, we make a special kind of copy of each form, and splice that into place. These copies will have each interesting symbol replaced with an uninterned symbol whose value cell contains the symbol it replaced and whose *section* property contains the section in which the original symbol occurred. We'll these uninterned symbols *referring symbols*.

First, we'll need a routine that does the substitution just described. The substitution is done blindly and without regard to the syntax or semantics of Common Lisp, since we can't walk pre-tangled code.

```
(defun substitute-symbols (form section &aux symbols)
  (labels ((get-symbols (form)
             (cond ((and (symbolp form)
                         (member (symbol-package form) *index-packages*))
                    (pushnew form symbols))
                   ((atom form) nil)
                   (t (get-symbols (car form))
                      (get-symbols (cdr form))))))
    (get-symbols form)
    (sublis (map 'list
                 (λ (sym)
                   (let ((refsym (make-symbol (symbol-name sym))))
                     (setf (symbol-value refsym) sym)
                     (setf (get refsym 'section) section)
                     (cons sym refsym)))
                 symbols)
            form)))
```

233. This next function goes the other way: given a symbol, it determines whether it is a referring symbol, and if so, it returns the symbol referred to and the section it came from. Otherwise, it just returns the given symbol. This interface makes it convenient to use in a *multiple-value-bind* form without having to apply a predicate first.

```
(defun symbol-provenance (symbol)
  (let ((section (get symbol 'section)))
    (if (and (null (symbol-package symbol))
             (boundp symbol)
             section)
      (values (symbol-value symbol) section)
      symbol)))
```

234. To get referring symbols in the tangled code, we'll use an *:around* method on *section-code* that conditions on a special variable, **indexing**, that we'll bind to true while we're tangling for the purposes of indexing.

We can't feed the raw section code to *substitute-symbols*, since it's not really Lisp code: it's full of markers and such. So we'll abuse the tangler infrastructure and use it to do marker replacement, but *not* named-section expansion.

```
(defmethod section-code :around ((section section))
  (let ((code (call-next-method)))
    (if *indexing*
        (substitute-symbols (tangle code :expand-named-sections nil) section)
        code)))
```

235. ⟨ Global variables 12 ⟩ +≡

```
(defvar *indexing* nil)
```

236. The top-level indexing routine will use this function to obtain the completely tangled code with referring symbols, and *that*'s what we'll walk.

```
(defun tangle-code-for-indexing (sections)
  (let ((*indexing* t))
    (tangle (unnamed-section-code-parts sections))))
```

237. Now we're ready to define a specialized walker that does the actual indexing.

```
(defclass indexing-walker (walker)
  ((index :accessor walker-index :initarg :index :initform (make-index))))
```

238. We have to override the walker's macro expansion function, since the forms that we're considering might be or contain referring symbols, which won't have macro definitions. There are two important cases here: (1) a form that is a referring symbol whose referent is a symbol macro in the current environment; and (2) a compound form, the operator of which is a referring symbol whose referent is a macro in the current environment. In both cases, we'll index the use of the (symbol) macro, then hand control off to the next method for the actual expansion.

```
(defmethod macroexpand-for-walk ((walker indexing-walker) form env)
  (typecase form
    (symbol
     (multiple-value-bind (symbol section) (symbol-provenance form)
       (cond (section
              (case (variable-information symbol env)
                (:symbol-macro
                 (index-variable (walker-index walker) symbol section env)
                 (call-next-method walker (cons symbol (cdr form)) env))
                (t form)))
             (t (call-next-method)))))
    (cons
     (multiple-value-bind (symbol section) (symbol-provenance (car form))
       (cond (section
              (case (function-information symbol env)
                (:macro
                 (index-funcall (walker-index walker) symbol section env)
                 (call-next-method walker (cons symbol (cdr form)) env))
                (t form)))
             (t (call-next-method)))))
    (t form)))
```

239. The only atoms we care about indexing are referring symbols. We'll return the referents; this is where the reverse-substitution of referring symbols takes place.

```
(defmethod walk-atomic-form ((walker indexing-walker) form env &optional evalp)
  (declare (ignore evalp))
  (if (symbolp form)
      (multiple-value-bind (symbol section) (symbol-provenance form)
        (when section
          (index-variable (walker-index walker) symbol section env))
        symbol)
      form))
```

240. We'll index all function-call-like forms whose *operator* is a referring symbol.

```
(defmethod walk-compound-form :before ((walker indexing-walker) operator form env)
  (declare (ignore form))
  (multiple-value-bind (symbol section) (symbol-provenance operator)
    (when section
      (index-funcall (walker-index walker) symbol section env))))
```

241. The walker methods for all of the function-defining and binding forms call down to this function, passing keyword arguments that describe the context.

```
(defmethod walk-function-name :before ((walker indexing-walker) function-name env &rest args &key def)
  (let ((index (walker-index walker)))
    (typecase function-name
      (symbol
       (multiple-value-bind (symbol section) (symbol-provenance function-name)
         (when section
           (if def
             (apply #'index-defun index symbol section :allow-other-keys t args)
             (index-funcall index symbol section env)))))
      (setf-function-name
       (multiple-value-bind (symbol section) (symbol-provenance (cadr function-name))
         (when section
           (if def
             (apply #'index-defun index `(setf ,symbol) section :allow-other-keys t args)
             (index-funcall index symbol section env))))))))
```

242. We'll treat *defun* and *defmacro* as special forms, since otherwise they will get macro-expanded before we get a chance to walk the function name. In fact, we won't even bother expanding them at all, since we don't care about the implementation-specific expansions, and we get everything we need from this simple walk.

Note in particular that we don't record the macro definition when we walk a *defmacro* form. For the macro definition to be available during the walk, the defining form must have been previously evaluated.

The indexing proper happens in the *walk-function-name* method we just defined, by way of *walk-lambda-expression*.

```
(macrolet ((define-defun-like-walker (operator)
             `(define-special-form-walker ,operator ((walker indexing-walker) form env)
                `(,(car form)
                  ,@(walk-lambda-expression walker (cdr form) env
                                            :operator (car form)
                                            :def t)))))
  (define-defun-like-walker defun)
  (define-defun-like-walker defmacro))
```

243. The special-variable-defining forms must also be walked as if they were special forms. Once again, we'll just skip the macro expansions.

```
(macrolet ((define-indexing-defvar-walker (operator)
             `(define-special-form-walker ,operator ((walker indexing-walker) form env)
                `(,(car form)
                  ,(multiple-value-bind (symbol section) (symbol-provenance (cadr form))
                     (when section
                       (index-defvar (walker-index walker) symbol section
                                     :operator (car form)
                                     :special t))
                     symbol)
                  ,@(walk-list walker (cddr form) env)))))
  (define-indexing-defvar-walker defvar)
  (define-indexing-defvar-walker defparameter)
  (define-indexing-defvar-walker defconstant))
```

244. The default method for *walk-declaration-specifiers* just returns the given specifiers. That's not sufficient for our purposes here, though, because we need to replace referring symbols with their referents; otherwise, the declarations would apply to the wrong symbols.

Walking general declaration expressions is difficult, mostly because of the shorthand notation for type declarations. Fortunately, we only need to care about *special* declarations, so we just throw everything else away.

```
(defmethod walk-declaration-specifiers ((walker indexing-walker) decls env)
   (loop for (identifier . data) in decls
         if (eql identifier 'special)
           collect `(special ,@(walk-list walker data env))))
```

245. Now we'll turn to the various CLOS forms. We'll start with a little macro to pull off method qualifiers from a *defgeneric* form or a method description. Syntactically, the qualifiers are any non-list objects preceding the specialized λ-list.

```
(defmacro pop-qualifiers (place)
   `(loop until (listp (car ,place)) collect (pop ,place)))
```

246. For *defgeneric* forms, we're interested in the name of the generic function being defined and any methods that may be specified as method descriptions.

```
(define-special-form-walker defgeneric ((walker indexing-walker) form env)
   (destructuring-bind (operator function-name lambda-list &rest options) form
      `(,operator
        ,(walk-function-name walker function-name env :operator 'defgeneric
                             :generic t
                             :def t)
        ,(walk-lambda-list walker lambda-list nil env)
        ,@(loop for form in options
                collect (case (car form)
                              (:method ⟨ Walk the method description in form 248 ⟩)
                              (t (walk-list walker form env)))))))
```

247. Method descriptions are very much like *defmethod* forms with an implicit function name; this routine walks both. The function name (if non-null) and qualifiers (if any) should have been walked already; we'll walk the specialized λ-list and body forms here.

```
(defun walk-method-definition (walker operator function-name qualifiers
                               lambda-list body env)
   (multiple-value-bind (body-forms decls doc) (parse-body body :walker walker :env env)
      (multiple-value-bind (lambda-list env) (walk-specialized-lambda-list walker lambda-list decls env)
         `(,operator
           ,@(when function-name `(,function-name))
           ,@qualifiers
           ,lambda-list
           ,@(if doc `(,doc))
           ,@(if decls `((declare ,@decls)))
           ,@(walk-list walker body-forms env)))))
```

248. ⟨ Walk the method description in *form* 248 ⟩ ≡

```
(let∗ ((operator (pop form))
       (qualifiers (mapcar (λ (q) (walk-atomic-form walker q env nil))
                           (pop-qualifiers form)))
       (lambda-list (pop form))
       (body form))
   ;; This is purely for its side-effect on the index.
   (walk-function-name walker function-name env
                       :operator 'defmethod
                       :generic t
                       :qualifiers qualifiers
                       :def t)
   (walk-method-definition walker operator nil qualifiers lambda-list body env))
```

This code is used in section 246.

249. Walking a *defmethod* form is almost, but not quite, the same as walking a method description.

```
(define-special-form-walker defmethod
      ((walker indexing-walker) form env &aux
       (operator (pop form))
       (function-name (pop form)) ; don't walk yet: wait for the qualifiers
       (qualifiers (mapcar (λ (q) (walk-atomic-form walker q env nil))
                           (pop-qualifiers form)))
       (lambda-list (pop form))
       (body form))
   (walk-method-definition walker operator
                           (walk-function-name walker function-name env
                                               :operator 'defmethod
                                               :generic t
                                               :qualifiers qualifiers
                                               :def t)
                           qualifiers lambda-list body env))
```

250. We'll walk *defclass* and *define-condition* forms in order to index the class names, super-classes, and accessor methods.

```
(macrolet ((define-defclass-walker (operator)
              `(define-special-form-walker ,operator ((walker indexing-walker) form env)
                 (destructuring-bind (operator name supers slot-specs &rest options) form
                   (multiple-value-bind (symbol section) (symbol-provenance name)
                     (when section
                       (add-index-entry (walker-index walker)
                                        (list symbol (make-sub-heading operator))
                                        section
                                        :def t))
                     `(,operator
                       ,(walk-atomic-form walker symbol env nil)
                       ,(mapcar (λ (super)
                                   ⟨ Index the use of the superclass super 251 ⟩)
                                 supers)
                       ,(mapcar (λ (spec) (walk-slot-specifier walker spec env)) slot-specs)
                       ,@(walk-list walker options env)))))))
  (define-defclass-walker defclass)
  (define-defclass-walker define-condition))
```

251. ⟨ Index the use of the superclass *super* 251 ⟩ ≡

```
(multiple-value-bind (symbol section) (symbol-provenance super)
  (when section
    (add-index-entry (walker-index walker)
                     (list symbol (make-sub-heading operator))
                     section))
  (walk-atomic-form walker symbol env nil))
```

This code is used in section 250.

252. The only slot options we care about are *:reader*, *:writer*, and *:accessor*. We index the methods implicitly created by those options.

```
(defun walk-slot-specifier (walker spec env)
  (etypecase spec
    (symbol (walk-atomic-form walker spec env nil))
    (cons (destructuring-bind (name &rest options) spec
            (loop for (opt-name opt-value) on options by #'cddr
                  if (member opt-name '(:reader :writer :accessor))
                    do ⟨ Index opt-value as a slot access method 253 ⟩)
            `(,(walk-atomic-form walker name env nil)
              ,@(walk-list walker options env))))))
```

253. ⟨ Index *opt-value* as a slot access method 253 ⟩ ≡

```
(multiple-value-bind (symbol section) (symbol-provenance opt-value)
  (when section
    (index-defun (walker-index walker) symbol section :operator 'defmethod :generic t)
    (when (eql opt-name :accessor)
      (index-defun (walker-index walker) `(setf ,symbol) section :operator 'defmethod :generic t))))
```

This code is used in section 252.

254. And here, finally, is the top-level indexing routine: it walks the tangled, symbol-replaced code of the given sections and returns an index of all of the interesting symbols so encountered.

```
(defun index-sections (sections &key
                                (index *index*)
                                (walker (make-instance 'indexing-walker :index index)))
  (let ((*evaluating* t))
    (dolist (form (tangle-code-for-indexing sections) (walker-index walker))
      (walk-form walker form))))
```

255. All that remains now is to write the index entries out to the index file.

```
(set-weave-dispatch 'index
  (λ (stream index)
    (maptree (λ (entry) (write entry :stream stream))
             (index-entries index))))
(set-weave-dispatch 'index-entry
  (λ (stream entry)
    (format stream "\\I~/clweb::print-entry-heading/~{,␣~W~}.~%"
            (entry-heading entry)
            (sort (copy-list (entry-locators entry)) #'<
                  :key (λ (loc) (section-number (location loc)))))))
(set-weave-dispatch 'section-locator
  (λ (stream loc)
    (format stream "~:[~D~;\\[~D]~]"
            (locator-definition-p loc)
            (section-number (location loc)))))
(defun print-entry-heading (stream heading &rest args)
  (declare (ignore args))
  (pprint-logical-block (stream heading)
    (pprint-exit-if-list-exhausted)
    (loop
      (let ((heading (pprint-pop)))
        (cond ((symbolp heading) (format stream "\\(~W\\)" heading))
              (t (format stream "~[\\.~;\\9~]{~/clweb::print-TeX/}"
                         (position (type-of heading) '(tt-heading custom-heading))
                         (read-tex-from-string (heading-name heading))))))
      (pprint-exit-if-list-exhausted)
      (write-char #\␣ stream))))
```

256. Index.

⟨ Accumulate limbo text in *commentary* 107 ⟩ Used in section 106.
⟨ Accumulate Lisp-mode material in *code* 110 ⟩ Used in section 106. Cited in section 114.
⟨ Accumulate TEX-mode material in *commentary* 109 ⟩ Used in section 106.
⟨ Build a bit vector from the characters in *bits* 81 ⟩ Used in section 80.
⟨ Build a logical block from *list* 145 ⟩ Used in section 144.
⟨ Call the standard reader macro function for #⟨*sub-char*⟩ 84 ⟩ Used in section 83.
⟨ Check for an ambiguous match, and raise an error in that case 28 ⟩ Used in section 26.
⟨ Check for *&whole* and *&environment* vars, and augment the lexical environment with them if found 189 ⟩ Used in section 188.
⟨ Choose an appropriate heading class for a method 220 ⟩ Used in section 219.
⟨ Complain about any unused named sections 124 ⟩ Used in section 121.
⟨ Complain about starting a section without a commentary part 111 ⟩ Used in section 110.
⟨ Condition classes 27, 38, 99, 112, 123, 179 ⟩ Used in section 5.
⟨ Extract the required parameters from *lambda-list* 198 ⟩ Used in section 197.
⟨ Find the tail of the list marker 63 ⟩ Used in section 62.
⟨ Finish the last section and initialize section variables 108 ⟩ Used in sections 106 and 111.
⟨ Global variables 12, 15, 29, 32, 35, 39, 46, 49, 57, 59, 73, 127, 128, 210, 225, 235 ⟩ Used in section 5.
⟨ Index the use of the superclass *super* 251 ⟩ Used in section 250. Cited in section 256.
⟨ Index *opt-value* as a slot access method 253 ⟩ Used in section 252.
⟨ Initialize global variables 13, 16, 30, 211, 226 ⟩ Used in sections 118, 121, and 125.
⟨ Insert a new node with key *item* and return it 21 ⟩ Used in section 20.
⟨ Maybe push the newline marker 113 ⟩ Used in section 110.
⟨ Merge defaults for tangler file names 122 ⟩ Used in section 121.
⟨ Merge defaults for weaver file names 126 ⟩ Used in section 125.
⟨ Process dotted rest var 196 ⟩ Used in section 188.
⟨ Process the symbol in *arg* as a parameter 195 ⟩ Used in sections 191, 192, 193, and 194.
⟨ Process *arg* as a keyword parameter 193 ⟩ Used in section 188.
⟨ Process *arg* as a required parameter 191 ⟩ Used in section 188.
⟨ Process *arg* as an auxiliary variable 194 ⟩ Used in section 188.
⟨ Process *arg* as an environment parameter 190 ⟩ Used in section 188.
⟨ Process *arg* as an optional parameter 192 ⟩ Used in section 188.
⟨ Read characters up to, but not including, the next newline 71 ⟩ Used in section 70.
⟨ Read the next object from *stream* and push it onto *list* 67 ⟩ Used in sections 65 and 66.
⟨ Read the next token from *stream*, which might be a consing dot 66 ⟩ Used in section 65.
⟨ Signal an error about section definition in Lisp mode 100 ⟩ Used in section 98.
⟨ Signal an error about section name use in TEX mode 101 ⟩ Used in section 98.
⟨ Trim whitespace and reverse *commentary* and *code* 114 ⟩ Used in section 106.
⟨ Update the section name if the new one is better 34 ⟩ Used in section 33.
⟨ Walk the method description in *form* 248 ⟩ Used in section 246. Cited in section 256.
⟨ Walker generic functions 168, 170, 172, 174, 178, 186 ⟩ Used in section 162.

CLWEB