1. Introduction. This is CLWEB, a literate programming system for Common Lisp by Alex Plotnick. 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).

2. A CLWEB source file, or *web*, consists of a mixture of TEX, Lisp, and WEB control codes. Which is primary depends on your point of view; the CWEB manual 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 also think of a web as a Lisp program with documentation blocks and special directives 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 web for compilation and evaluation by a machine, and the weaver pretty-prints it for reading by a human. These operations reflect the two uses of a literate program, and the two audiences by whom it is to 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 loaded into a Lisp image with *load*. The function *load-web* is analogous to *load*: it loads a web into the Lisp environment without first tangling and compiling it. (There's also *load-sections-from-temp-file*, which 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 should generally never call it directly.) Tangling can also produce and compile a *tests file*, which contains any tests defined alongside the main program as part of the web.

The weaver has a single entry point: *weave* takes a web as input and generates a TEX file containing a pretty-printed version of it, along with a complete index of the classes, functions, variables, *&c.* defined by the program.

4. We begin by defining a package for the CLWEB system. In addition to the top-level tangling and weaving functions, we export a handful of public utility functions, some global variables that control operations of the tangler and weaver, and condition classes for errors and warnings that may be signaled while processing a web. There's also the pair of symbols *clweb-file* and *weave-op*, which are not used for anything in this program, but are used to name ASDF operations in the optional companion web *asdf-operations.clw*.

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  #+:sbcl (require "SB-CLTL2"))
(defpackage "CLWEB"
  (:documentation "A␣literate␣programming␣system␣for␣Common␣Lisp.")
  (:use "COMMON-LISP")
  (:export "TANGLE-FILE"
           "LOAD-WEB"
           "WEAVE"
           "LOAD-SECTIONS-FROM-TEMP-FILE"
           "INPUT-FILE-PATHNAME"
           "LISP-FILE-PATHNAME"
           "TEX-FILE-PATHNAME"
           "INDEX-FILE-PATHNAME"
           "SECTIONS-FILE-PATHNAME"
           "FASL-FILE-PATHNAME"
           "TESTS-FILE-PATHNAME"
           "TANGLE-FILE-PATHNAMES"
           "WEAVE-PATHNAMES"
           "*WEB-PATHNAME-DEFAULTS*"
           "*LISP-PATHNAME-DEFAULTS*"
           "*TEX-PATHNAME-DEFAULTS*"
           "*INDEX-PATHNAME-DEFAULTS*"
           "*SECTIONS-PATHNAME-DEFAULTS*"
           "*TESTS-FILE-PATHNAME-FUNCTION*"
           "*COMPILE-TESTS-FILE*"
           "*TANGLE-FILE-PATHNAME*"
           "*TANGLE-FILE-TRUENAME*"
           "*WEAVE-PRINT*"
           "*WEAVE-VERBOSE*"
           "*INDEX-LEXICAL-VARIABLES*"
           "AMBIGUOUS-PREFIX-ERROR"
           "SECTION-NAME-CONTEXT-ERROR"
           "SECTION-NAME-USE-ERROR"
           "SECTION-NAME-DEFINITION-ERROR"
           "UNUSED-NAMED-SECTION-WARNING"
           "CLWEB-FILE" "WEAVE-OP") ; see asdf-operations
  #+(:or :sbcl :ccl :allegro)
  (:import-from #+sbcl "SB-CLTL2" #+ccl "CCL" #+allegro "SYS"
                #-allegro "FUNCTION-INFORMATION"
                #-allegro "VARIABLE-INFORMATION"
                #-allegro "PARSE-MACRO"
                "AUGMENT-ENVIRONMENT")
  #+:sbcl
  (:import-from "SB-INT" "NAMED-LAMBDA"))
(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 36 ⟩

6. Next we'll define some utility functions and macros that we'll use throughout the program. Our first function simply treats its argument as a list designator.

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

7. This auxiliary function—shamelessly stolen from CLtL-2, Appendix C—is like *mapcar* but has two extra purposes: (1) it handles dotted lists; (2) it tries to make the result share with the argument *x* as much as possible.

```
(defun maptree (fn x)
  (if (atom x)
      (funcall fn x)
      (let ((a (funcall fn (car x)))
            (d (maptree fn (cdr x))))
        (if (and (eql a (car x)) (eql d (cdr x)))
            x
            (cons a d)))))
```

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

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

9. When we're accumulating text, we usually won't want to bother with empty strings. In such cases we'll 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 (temp (gensym)))
  `(let ((,temp ,obj))
     (when (if (stringp ,temp) (plusp (length ,temp)) ,temp)
       (push ,temp ,place))))
```

10. Our last utility macro goes by many names. Paul Graham calls it *with-gensyms*, but we'll use the more descriptive *with-unique-names*. It captures a common idiom used in writing macros: given a list of symbols, it executes the body in an environment augmented with bindings for each of those symbols to a fresh, uninterned symbol.

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (defmacro with-unique-names (symbols &body body)
    `(let ,(loop for symbol in symbols collect `(,symbol (copy-symbol ',symbol)))
       ,@body)))
```

11. File names. In TEX, '`\input foo`' reads tokens from the file named '`foo.tex`'. In Common Lisp, (*compile-file* `"foo"`) compiles the file named '`foo.lisp`' (or an implementation-dependent variant thereof). Those are just defaults, of course; '`\input foo.bar`' causes TEX to read tokens from '`foo.bar`', and similarly (*compile-file* `"foo.bar"`) compiles Lisp forms from '`foo.bar`'. In both systems, the type components of input filenames default in reasonable ways when omitted, but types that the user supplies are respected.

Output filenames are defaulted from the input filenames. For instance, '`\input foo`' will cause TEX to write a device-independent file named '`foo.dvi`' as output, along with '`foo.log`'. We will say that TEX implements the *default filename mapping*

$$\texttt{\textbackslash input foo : foo.tex} \rightarrow \begin{cases} \texttt{foo.dvi} \\ \texttt{foo.log} \end{cases}$$

Similarly, compiling a Lisp source file named '`foo.lisp`' produces by default an output file named '`foo.fasl`' (or an implementation-dependent variant thereof). So the Common Lisp file compiler implements the default filename mapping

$$(\textit{compile-file}\ \texttt{"foo"})\ \texttt{: foo.lisp} \rightarrow \{\, \texttt{foo.fasl}$$

(or an implementation-dependent variant thereof).

`CLWEB` implements the default filename mappings

$$(\textit{tangle-file}\ \texttt{"foo"})\ \texttt{: foo.clw} \rightarrow \begin{cases} \texttt{foo.fasl} \\ \texttt{foo.lisp} \\ \texttt{foo-tests.fasl} \\ \texttt{foo-tests.lisp} \end{cases}$$

and

$$(\textit{weave}\ \texttt{"foo"})\ \texttt{: foo.clw} \rightarrow \begin{cases} \texttt{foo.tex} \\ \texttt{foo.idx} \\ \texttt{foo.scn} \end{cases}$$

Filename defaulting is performed by the pathname manipulation functions that follow. We define a defaulting function for each type of file that `CLWEB` operates on, and one 'main' routine for each of *tangle-file* and *weave* (*tangle-file-pathnames* and *weave-pathnames*, respectively) that returns a fully defaulted set of filenames that the corresponding operation will write into. Thanks to Richard M. Kreuter for his careful analysis of the requirements for these routines.

12. The type componenents of the various input and output files take their defaults from the following global variables. We use a tiny helper function to make them so that their version and host parameters are set correctly.

⟨ Global variables 12 ⟩ ≡
```
(defvar *physical-pathname-host*
  #+:sbcl ""
  #-:sbcl :unspecific)
(defun make-default-pathname (type)
  (make-pathname :type type
                 :host *physical-pathname-host*
                 :version :newest))
(defvar *web-pathname-defaults* (make-default-pathname "clw"))
(defvar *lisp-pathname-defaults* (make-default-pathname "lisp"))
(defvar *tex-pathname-defaults* (make-default-pathname "tex"))
(defvar *index-pathname-defaults* (make-default-pathname "idx"))
(defvar *sections-pathname-defaults* (make-default-pathname "scn"))
```
See also sections 17, 24, 27, 42, 46, 49, 50, 55, 62, 65, 76, 145, 146, 147, 158, 160, 164, 183, 287, 303, 308, 315, and 322.
This code is used in section 5.

13. All of the filename defaulting code eventually calls down to either *input-file-pathname* or *output-file-pathname*. Both return a fully defaulted, untranslated pathname.

```
(defun input-file-pathname (input-file)
  (let ((defaults (merge-pathnames *web-pathname-defaults*)))
    (merge-pathnames input-file defaults)))
(defun output-file-pathname (input-file &key output-file (defaults *default-pathname-defaults*))
  (assert (or (and output-file (pathname-type output-file))
              (pathname-type defaults))
          (defaults)
          "Missing␣default␣output␣file␣type.")
  (let* ((input-file (input-file-pathname input-file))
         (defaults (merge-pathnames (make-pathname :type nil :version :newest :defaults input-file) defaults)))
    (if output-file
        (merge-pathnames output-file defaults)
        defaults)))
```

14. Now we can define pathname defaulting routines for each of the output file types produced by `CLWEB`. Most of them are trivial wrappers around *output-file-pathname* that simply provide the appropriate defaults.

```
(macrolet ((defdefaults (name (input) defaults)
             (with-unique-names (args)
               `(defun ,name (,input &rest ,args)
                  (apply #'output-file-pathname ,input :defaults ,defaults ,args)))))
  (defdefaults lisp-file-pathname (input-file) *lisp-pathname-defaults*)
  (defdefaults tex-file-pathname (input-file) *tex-pathname-defaults*)
  (defdefaults index-file-pathname (output-file) *index-pathname-defaults*)
  (defdefaults sections-file-pathname (index-file) *sections-pathname-defaults*))
```

15. The tangler's primary output is a compiled Lisp file whose type is implementation-defined. So we must ask the file compiler where it would write its output if given a defaulted input filename.

```
(defun fasl-file-pathname (input-file &rest args)
  (apply #'compile-file-pathname (lisp-file-pathname input-file) :allow-other-keys t args))
```

16. Both *tangle-file* and *weave* take a *tests-file* argument with especially hairy defaulting behavior. If it's supplied as *nil*, no tests file will be written. If it's supplied and non-*nil*, it is defaulted from the output filename. And if it's unsupplied, the function stored in ∗*tests-file-pathname-function*∗ is called to construct a new filename from the input and output filenames.

```
(defun tests-file-pathname (input-file &key output-file (tests-file nil tests-file-supplied) &allow-other-keys)
  (if tests-file
      (output-file-pathname input-file :output-file tests-file :defaults output-file)
      (unless tests-file-supplied (funcall ∗tests-file-pathname-function∗ input-file output-file))))
```

17. The default tests filename function appends the string `"-tests"` to the output file's name and defaults everything else.

```
⟨ Global variables 12 ⟩ +≡
(defvar ∗tests-file-pathname-function∗
  (λ (input-file output-file &aux
      (output-file (pathname output-file))
      (tests-file (make-pathname :name (concatenate 'string (pathname-name output-file) "-tests")
                                 :version :newest
                                 :defaults output-file)))
    (output-file-pathname input-file
                          :output-file tests-file
                          :defaults output-file)))
```

18. And now we come to our final filename defaulting functions: *tangle-file-pathnames* returns the defaulted output filenames for *tangle-file*, and *weave-pathnames* returns the defaulted output files for *weave*. The conditionals are for supplied but null output filename arguments, which disable writing of the corresponding outputs; e.g., the tests or index files.

```
(defun tangle-file-pathnames (input-file &rest args &key output-file tests-file &allow-other-keys)
  "Compute␣and␣return␣the␣defaulted␣names␣of␣the␣tangler␣output␣files."
  (declare (ignorable output-file tests-file))
  (let∗ ((input-file (input-file-pathname input-file))
         (output-file (apply #'fasl-file-pathname input-file args))
         (lisp-file (lisp-file-pathname output-file))
         (tests-file (apply #'tests-file-pathname input-file :output-file lisp-file args))
         (tests-output-file (and tests-file (fasl-file-pathname tests-file))))
    (values output-file
            lisp-file
            tests-output-file
            tests-file)))

(defun weave-pathnames (input-file &key output-file (index-file nil index-file-supplied) &allow-other-keys)
  "Compute␣and␣return␣the␣defaulted␣names␣of␣the␣weaver␣output␣files."
  (let∗ ((input-file (input-file-pathname input-file))
         (output-file (tex-file-pathname input-file :output-file output-file))
         (index-file (unless (and index-file-supplied (not index-file))
                       (index-file-pathname output-file :output-file index-file)))
         (sections-file (and index-file (sections-file-pathname index-file))))
    (values output-file
            index-file
            sections-file)))
```

19. 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.

Finally, the section keeps track of where it was read, in a slot called *source-location*.

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)
   (source-location :reader section-source-location :initarg :source-location))
  (:default-initargs :name nil :commentary nil :code nil :source-location nil))
```

20. Sections introduced with `@*` ('starred' sections) begin a new 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.

Starred sections have a numeric parameter associated with them called 'depth'. The depth of a starred section affects how it is formatted in the woven output: the section name is indented according to its depth in the table of contents, and sections with small depths (i.e., close to 0) force a page break.

The tangler makes no distinction at all between sections with stars and ones with none upon thars.

```
(defclass starred-section (section)
  ((depth :reader section-depth :initarg :depth))
  (:default-initargs :depth 0))
(defun starred-section-p (object) (typep object 'starred-section))
```

21. Non-starred sections have no depth, but it shouldn't be an error to ask.

```
(defmethod section-depth ((section section)) nil)
```

22. Sections that begin with `@t` are *test sections*. They are used to include test cases alongside the program, and 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 nil)))
(defclass starred-test-section (test-section starred-section) ())
(defun test-section-p (object) (typep object 'test-section))
(defmethod initialize-instance :after ((section test-section) &key)
  (when (> (fill-pointer *sections*) 0)
    (setf (test-for-section section) (elt *sections* (1- (fill-pointer *sections*))))))
```

23. 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) ())
```

24. 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)
```

25. ⟨ Initialize global variables 25 ⟩ ≡

```
(setf (fill-pointer *sections*) 0)
(setf *current-section* nil)
```

See also sections 28, 43, 288, 304, and 316.

This code is used in sections 143, 150, and 157.

26. 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) &key)
  (setq *current-section* (push-section section)))
```

27. 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))
```

28. ⟨ Initialize global variables 25 ⟩ +≡
```
(setf (fill-pointer *test-sections*) 0)
```

29.
```
(defmethod push-section ((section test-section))
  (let ((*sections* *test-sections*))
    (call-next-method)))
```

30. 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*))
```

31. 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))
```

32. 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 33 ⟩
                 (values nil nil)))))))
```

33. ⟨ Insert a new node with key *item* and return it 33 ⟩ ≡

```
(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 32.

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

```
(defgeneric map-bst (function tree))
(defmethod map-bst (function (tree null))
  (declare (ignore function)))
(defmethod map-bst (function (tree binary-search-tree))
  (map-bst function (left-child tree))
  (funcall function tree)
  (map-bst function (right-child tree)))
```

35. 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 are 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))
  (let ((sections (named-section-sections section)))
    (when (null sections)
      (error 'undefined-named-section-error
             :format-control "Undefined␣named␣section␣<~A>."
             :format-arguments (list (section-name section))))
    (section-number (first sections))))
```

36. ⟨ Condition classes 36 ⟩ ≡

```
(define-condition undefined-named-section-error (simple-error) ())
```

See also sections 40, 53, 126, 138, and 153.

This code is used in section 5.

37. 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))))
```

38. 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))))))
```

39. 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 41 ⟩
            (values node t))
        (values node nil))))
```

40. ⟨ Condition classes 36 ⟩ +≡
```
(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)))))
```

41. 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 41 ⟩ ≡
```
(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 39.

42. 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)
```

43. ⟨ Initialize global variables 25 ⟩ +≡
```
(setq ∗named-sections∗ nil)
```

44. 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. Thanks to Richard M. Kreuter for the basic structure of this implementation.

```
(defun squeeze (string)
  (with-input-from-string (*standard-input* string)
    (with-output-to-string (*standard-output*)
      (handler-case
          (flet ((snarf-whitespace ()
                   (loop for char = (read-char)
                         while (whitespacep char)
                         finally (unread-char char))))
            (loop initially (snarf-whitespace)
                  for char = (read-char)
                  do (write-char (if (whitespacep char)
                                     (progn (snarf-whitespace) #\␣)
                                     char))))
        (end-of-file ())))))
```

45. The predicate *whitespacep* determines whether or not a given character should be treated as whitespace. Note, however, that this routine does not—and can not, at least not portably—examine the current readtable to determine which characters currently have $whitespace_2$ syntax.

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

46. Only the characters named ‘Newline’ and ‘Space’ are required to be present in a conforming Common Lisp implementation, but most also support the semi-standard names ‘Tab’, ‘Linefeed’, ‘Return’, and ‘Page’. Any of these that are supported should be considered whitespace characters.

```
⟨ Global variables 12 ⟩ +≡
(defparameter *whitespace*
  #.(coerce (remove-duplicates
             (remove nil (mapcar #'name-char
                                 '("Newline" "Space" "Tab" "Linefeed" "Return" "Page"))))
            'string))
```

47. 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*)
      (setq *named-sections* (make-instance 'named-section :name name))
      (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 48 ⟩)
        section)))
```

48. 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 48 ⟩ ≡

```
(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 47.

49. Reading a web. We distinguish five distinct modes for reading. *Limbo mode* is used for TEX text that precedes the first section in a web. *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 *restricted mode* is used for reading material in section names and a few other places. The modes are named by the symbols *:limbo*, *:tex*, *:lisp*, *:inner-lisp*, and *:restricted*, respectively.

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

50. We use separate readtables for each mode, stored as an alist keyed on the mode name. We add an extra mode with name *nil* that keeps a virgin copy of the standard readtable for when we want to read in 'no-mode'. Although we will modify the readtables stored there, the **readtables** list itself is never changed.

⟨ Global variables 12 ⟩ +≡
```
(defvar *readtables* (loop for mode in (cons 'nil *modes*) collect (cons mode (copy-readtable nil))))
```

51. We'll access the mode-specific readtables via *readtable-for-mode*.

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

52. The following macro executes the given *body* with **readtable** bound appropriately for *mode*.

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

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

⟨ Condition classes 36 ⟩ +≡
```
(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)))))
```

54. This function just signals an error of type *simple-reader-error*.

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

55. We'll use this object as our end-of-file marker (i.e., 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"))
```

56. But instead of using the variable ∗*eof*∗ directly in calls to *read*, we'll use the symbol macro *eof*, which bypasses the variable lookup at run-time.

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (define-symbol-macro eof (load-time-value *eof* t)))
```

57. We'll test for our EOF value using *eof-p*.

```
(defun eof-p (object) (eq object eof))
(deftype eof () '(satisfies eof-p))
```

58. 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)))))
```

59. We need to be careful about reader macro functions that don't return any values. (In standard ANSI Common Lisp syntax, the only reader macros that might return zero values are those for the two comment syntaxes.) For example, if the input file contains '`(#|...|#)`' and we blindly call *read* in the reader macro function for '#\(', it will skip over the comment and invoke the reader macro function for '#\)', which will signal an error.

Our solution is to encapsulate any reader macro function that might return zero values with a closure that throws to our gatekeeper entry point to the reader, *read-maybe-nothing-internal*. That function (and thus also its two front-ends *read-maybe-nothing* and *read-maybe-nothing-preserving-whitespace*) returns a list containing either zero or one values.

```
(defun wrap-reader-macro-function (function)
  (λ (&rest args)
    (let ((values (multiple-value-list (apply function args))))
      (assert (null (cdr values))
              (values)
              "Reader␣macro␣function␣~S␣returned␣more␣than␣one␣value." function)
      (if values
          (values-list values)
          (throw 'read-nothing (values))))))

(defun read-maybe-nothing-internal (read stream eof-error-p eof-value recursive-p)
  (multiple-value-list
   (catch 'read-nothing
     (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))
```

60. Tracking character position. 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. It's these proxy streams that we'll pass around, so all the standard stream functions will still work.

```
(defclass charpos-stream ()
  ((charpos :initarg :charpos :reader charpos :initform 0)
   (lineno :initarg :lineno :reader lineno :initform 1)
   (proxy-stream :accessor charpos-proxy-stream :initarg :proxy)))
```

61. 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))
(defmethod charpos :before ((stream charpos-stream))
  (update-charpos-stream-positions stream))
(defmethod lineno ((stream stream)) nil)
(defmethod lineno :before ((stream charpos-stream))
  (update-charpos-stream-positions stream))
(defun update-charpos-stream-positions (stream)
  (with-slots (charpos lineno) stream
    (loop for char across (get-charpos-stream-buffer stream)
          do (case char
               (#\Tab (incf charpos (- ∗tab-width∗ (rem charpos ∗tab-width∗))))
               (#\Newline (incf lineno) (setf charpos 0))
               (t (incf charpos))))))
```

62. ⟨ Global variables 12 ⟩ +≡

```
(defparameter ∗tab-width∗ 8)
```

63. 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 ((instance charpos-input-stream) slot-names &rest initargs &key stream &aux
                              (element-type (stream-element-type stream)))
  (apply #'call-next-method instance slot-names
         :proxy (make-echo-stream stream (make-string-output-stream :element-type element-type))
         initargs))
(defmethod get-charpos-stream-buffer ((stream charpos-input-stream))
  (get-output-stream-string (echo-stream-output-stream (charpos-proxy-stream stream))))
```

64. 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 ((instance charpos-output-stream) slot-names &rest initargs &key stream &aux
                              (element-type (stream-element-type stream)))
  (apply #'call-next-method instance slot-names
         :proxy (make-broadcast-stream (make-string-output-stream :element-type element-type) stream)
         initargs))
(defmethod get-charpos-stream-buffer ((stream charpos-output-stream))
  (get-output-stream-string (first (broadcast-stream-streams (charpos-proxy-stream stream)))))
```

65. 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))
```

66.

```
(defmethod initialize-instance :after ((instance charpos-stream) &key)
  (setf (gethash (charpos-proxy-stream instance) *charpos-streams*) instance))
```

67. 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 stream-lineno (stream)
  (lineno (or (gethash stream *charpos-streams*)
              (error "Not␣tracking␣lineno␣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)))))
```

68. 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))
```

69. 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))))
```

70. Stream utilities. 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 bound to a local function that 'rewinds' that stream to its state prior to any reads that were executed in the body. If *rewind* is invoked more than once, each subsequent invocation will rewind to the state just after the previous one.

```
(defmacro with-rewind-stream ((var stream &optional (rewind 'rewind)) &body body)
  (with-unique-names (in out close)
    `(let* ((,out (make-string-output-stream))
            (,var (make-echo-stream ,stream ,out))
            (,close (list ,var ,out)))
       (flet ((,rewind ()
                (let ((,in (make-string-input-stream (get-output-stream-string ,out))))
                  (prog1 (setq ,var (make-concatenated-stream ,in ,var))
                    (pushnew ,var ,close)
                    (pushnew ,in ,close)))))
         (declare (ignorable (function ,rewind)))
         (unwind-protect (progn ,@body)
           (map nil #'close ,close))))))
```

71. 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.

```
(defmacro read-with-echo ((stream object echoed) &body body)
  (with-unique-names (out echo raw-output length char)
    `(with-open-stream (,out (make-string-output-stream))
       (with-open-stream (,echo (make-echo-stream ,stream ,out))
         (let* ((,object (read-preserving-whitespace ,echo))
                (,raw-output (get-output-stream-string ,out))
                (,length (length ,raw-output))
                (,echoed (subseq ,raw-output 0 (if ⟨ Should we include the last character of raw-input? 72 ⟩
                                                   ,length
                                                   (1- ,length)))))
           (declare (ignorable ,object ,echoed))
           ,@body)))))
```

72. We have to be very careful here about delimiters. For self-delimiting forms like quoted strings, the final delimiter will be—and should be—the last character consumed. But if there's some other kind of delimiter—a closing parenthesis, say—then it should *not* be delivered as part of *echoed*. The problem is that such a delimiter may very well have been read and then unread, and the standard says that when *unread-char* is called with an echo stream as its *input-stream* argument, "no attempt is made to undo any echoing of the character that might already have been done on *input-stream*. However," it continues, "characters placed on *input-stream* by *unread-char* are marked in such a way as to inhibit later re-echo by *read-char*." So we can detect a previously-unread character by reading and then unreading the next character in the echo stream and seeing if it actually gets echoed.

⟨ Should we include the last character of *raw-input*? 72 ⟩ ≡

```
(let ((,char (read-char ,echo nil eof)))
  (or (eof-p ,char) ; clearly 'yes' for EOF
      (progn (unread-char ,char ,echo)
             (plusp (length (get-output-stream-string ,out))))))
```

This code is used in section 71.

73. Markers. 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 (object) (typep object 'marker))
(defgeneric marker-boundp (marker))
(defmethod marker-boundp ((marker marker))
  (slot-boundp marker 'value))
```

74. We'll define specialized pretty-printing routines for some of the objects we read for use by the tangler. Since they depend heavily on the specific representations, they're best defined alongside the readers. A few of these will also be used by the weaver, so we'll define the accessor for that dispatch table, too.

⟨ Define the pprint dispatch table setters 148 ⟩

75. Here's a simple pretty-printing routine that 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.

```
(set-tangle-dispatch 'marker
  (λ (stream marker)
    (when (marker-boundp marker)
      (write (marker-value marker) :stream stream))))
```

76. 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)
```

77. 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. We'll interpret two newlines in a row as ending a paragraph, as in TEX; the weaver will insert a bit of extra vertical space when it encounters such markers.

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)))
(defclass par-marker (newline-marker) ())
(defun newlinep (obj) (typep obj 'newline-marker))
(set-macro-character #\Newline
  (λ (stream char)
    (declare (ignore char))
    (case (peek-char nil stream nil eof t)
      (#\Newline (read-char stream t nil t)
                 (make-instance 'par-marker))
      (otherwise (make-instance 'newline-marker))))
  nil (readtable-for-mode :lisp))
```

78. 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).

79. *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 80 ⟩)
           (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))))))
```

80. 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 80 ⟩ ≡

```
(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 79.

81. 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))
```

82. 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 83 ⟩
        else
          do ⟨ Read the next object from stream and push it onto list 84 ⟩
        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))
```

83. 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 83 ⟩ ≡
(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."))
           (unread-char following-char stream)
           (push *consing-dot* list)
           (push charpos charpos-list))
          (t (rewind)
             ⟨ Read the next object from stream and push it onto list 84 ⟩)))))
```

This code is used in section 82.

84. 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* 84 ⟩ ≡
```
(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 82 and 83.

85. *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)))
```

86. *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)))
```

87. 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 88 ⟩)))
(set-macro-character #\; (wrap-reader-macro-function #'comment-reader)
                     nil (readtable-for-mode :lisp))
```

88. ⟨ Read characters up to, but not including, the next newline 88 ⟩ ≡
```
(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 87.

89. *Backquote* has formally defined semantics, but the representation is explicitly implementation-specific. We therefore need to supply our own backquote reader to ensure a consistent representation across implementations.

This backquote implementation is based on Appendix C of CLtL-2. It has been modified to conform to the overall style of this program, to support commas inside vectors, and to remove the code simplifier. This last is in the interest of simplicity: because we preserve backquotes during tangling, we can leave optimization to the Lisp implementation.

90. We use conses to represent backquoted forms (since, as we'll see, backquote is an ordinary macro), but we'll use atomic instances of a dedicated class to represent commas. The reason is that we depend on the pretty printer to correctly print backquoted forms, and even though most Lisp implementations support printing their own representations of backquote and comma, there's no reason to expect them to correctly support ours. (They don't even get their own representations right all of the time.) Moreover, providing a pretty printing method for conses that have some specialized comma representation as their car is not sufficient, because many implementation-provided pretty printing routines (e.g., for *defun*, *let*, *&c.*) blindly call down to something like *pprint-fill* without first distinguishing between atoms and lists. (Thanks to Duane Rettig of Franz, Inc. for helping to track this down.) Using an atomic representation guarantees that our commas won't accidentally be printed as lists.

It is still possible for this scheme to go wrong, though: if an implementation-supplied pretty printing routine expects a list in some context and gets an atom instead, it could signal an error. If this happens to you, please report it to the author.

```
(defvar *backquote* (make-symbol "BACKQUOTE"))
(defun backquotep (object) (eq object *backquote*))
(deftype backquote-form () '(cons (satisfies backquotep)))

(defclass comma ()
  ((form :initarg :form :initform nil)))
(defclass splicing-comma (comma)
  ((modifier :reader comma-modifier :initarg :modifier :type character)))

(defmethod comma-modifier ((comma comma)) nil)

(defun make-comma (modifier form)
  (if modifier
      (make-instance 'splicing-comma :modifier modifier :form form)
      (make-instance 'comma :form form)))

(defun commap (obj) (typep obj 'comma))
```

91. To process a comma, we need to tangle the form being unquoted. If that form is a named section reference, we take the car of the tangled form, on the assumption that you can't meaningfully unquote more than one form. We accept an optional argument to disable the tangling for the sake of the weaver, which wants the untangled form.

```
(defgeneric comma-form (comma &key tangle))
(defmethod comma-form ((comma comma) &key (tangle t))
  (let ((form (slot-value comma 'form)))
    (unless tangle (return-from comma-form form))
    (let ((tangled-form (tangle form)))
      (typecase form
        (named-section
         (when (rest tangled-form)
           (cerror "Ignore␣the␣extra␣forms."
                   "Tried␣to␣unquote␣more␣than␣one␣form␣from␣section␣@<~A@>."
                   (section-name form)))
         (first tangled-form))
        (t tangled-form)))))
```

92. The reader macro functions for backquote and comma are straightforward.

```
(dolist (mode '(:lisp :inner-lisp))
  (set-macro-character #\‘
    (λ (stream char)
      (declare (ignore char))
      (list *backquote* (read stream t nil t)))
    nil (readtable-for-mode mode))
  (set-macro-character #\,
    (λ (stream char)
      (declare (ignore char))
      (case (peek-char nil stream t nil t)
        ((#\@ #\.) (make-comma (read-char stream t nil t)
                               (read stream t nil t)))
        (otherwise (make-comma nil (read stream t nil t)))))
    nil (readtable-for-mode mode)))
```

93. *#:backquote* is an ordinary macro (not a read-macro) that processes the expression *x*, looking for embedded commas. We use an uninterned symbol for *#:backquote* to avoid problems with macro expansion during indexing; the indexing code walker breaks symbol identity during macro expansion, but not for uninterned symbols.

```
(defmacro backquote (x)
  (bq-process x))
(setf (macro-function *backquote*) (macro-function 'backquote))
(defun bq-process (x &aux (x (tangle x)))
  (typecase x
    (simple-vector `(apply #'vector ,(bq-process (coerce x 'list))))
    (splicing-comma (error ",~C~S␣after␣‘" (comma-modifier x) (comma-form x)))
    (comma (comma-form x))
    (atom `(quote ,x))
    (backquote-form (bq-process (bq-process (cadr x))))
    (t ⟨ Process the list x for backquotes and commas 94 ⟩)))
```

94. We do one simplification here which, although not strictly in accordance with the formal rules on pages 528–529 of CLtL-2 (section 2.4.6 of ANSI Common Lisp), is necessary in the presence of nested backquotes; viz., we will never append *'nil* to the end of a list. This seems to be an error in the formal rules: in particular, reducing the case of a *nil*-terminated list to the general case of a dotted list appears to be overly simplistic.

⟨ Process the list *x* for backquotes and commas 94 ⟩ ≡

```
(do ((p x (cdr p))
     (q '() (cons (bracket (car p)) q)))
    ((and (atom p) (not (commap p)))
     (cons 'append (nreconc q (and p (list (list 'quote p))))))
  (typecase p
    (splicing-comma (error "Dotted␣,~C~S" (comma-modifier p) (comma-form p)))
    (comma (return (cons 'append (nreconc q (list (comma-form p))))))))
```

This code is used in section 93.

95. This implements the bracket operator of the formal rules.

```
(defun bracket (x)
  (typecase x
    (splicing-comma (comma-form x))
    (comma `(list ,(comma-form x)))
    (t `(list ,(bq-process x)))))
```

96. During tangling, we print backquotes and commas using the backquote syntax, as recommended (but not required) by section 2.4.6.1 of the ANSI standard.

```
(set-tangle-dispatch 'backquote-form
  (λ (stream obj) (format stream "`~W" (cadr obj))))
(set-tangle-dispatch 'comma
  (λ (stream obj) (format stream ",~@[~C~]~W" (comma-modifier obj) (comma-form obj))))
```

97. *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.

98. 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)))
```

99. 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 :reader simple-vector-marker-elements :initarg :elements)
   (element-type :reader simple-vector-marker-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))
  (let ((elements (tangle (simple-vector-marker-elements marker)))
        (element-type (simple-vector-marker-element-type marker)))
    (if (and elements (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))))
         (elements (or list *empty-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 elements))
          (make-instance 'simple-vector-marker :elements elements)))))

(dolist (mode '(:lisp :inner-lisp))
  (set-dispatch-macro-character #\# #\( #'simple-vector-reader (readtable-for-mode mode)))
```

100. Sharpsign asterisk also creates a vector, but the token following the asterisk must be composed entirely of the characters '0' and '1', which it uses to construct a *simple-bit-vector*. It supports the same kind of length abbreviation that `#()` does.

```
(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))
  (apply #'make-instance 'bit-vector-marker
         :elements (coerce ⟨ Read '0's and '1's from stream 101 ⟩ 'bit-vector)
         (when arg `(:length ,arg))))

(dolist (mode '(:lisp :inner-lisp))
  (set-dispatch-macro-character #\# #\* #'simple-bit-vector-reader (readtable-for-mode mode)))
```

101. ⟨ Read '0's and '1's from *stream* 101 ⟩ ≡

```
(loop for char = (read-char stream nil nil t)
      while (and char (or (char= char #\0) (char= char #\1)))
      collect (ecase char (#\0 0) (#\1 1))
      finally (when char (unread-char char stream)))
```

This code is used in section 100.

102. 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 pretty-printing 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)))
(set-tangle-dispatch 'read-time-eval
  (λ (stream obj)
    (format stream "#.~W" (read-time-eval-form obj))))
(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 ((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 (tangle form))))))
(dolist (mode '(:lisp :inner-lisp))
  (set-dispatch-macro-character #\# #\. #'sharpsign-dot-reader (readtable-for-mode mode)))
```

103. 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⟩ 104 ⟩))
(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))))
```

104. ⟨ Call the standard reader macro function for #⟨*sub-char*⟩ 104 ⟩ ≡

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

This code is used in section 103.

105. Sharpsign C reads a list of two real numbers representing, respectively, the real and imaginary parts of a complex number.

```
(defclass complex-marker (marker)
  ((components :reader complex-components :initarg :components)))
(defmethod marker-boundp ((marker complex-marker)) t)
(defmethod marker-value ((marker complex-marker))
  (apply #'complex (tangle (complex-components marker))))
(defun complex-reader (stream sub-char arg)
  (declare (ignore sub-char arg))
  (make-instance 'complex-marker
                 :components (read stream t nil t)))
(dolist (mode '(:lisp :inner-lisp))
  (set-dispatch-macro-character #\# #\C #'complex-reader (readtable-for-mode mode)))
```

106. Sharpsign nA constructs an n-dimensional array. We don't need any particularly special handling, but we have to override it anyway because the standard reader will be confused by finding markers where it expects sequences.

```
(defclass array-marker (marker)
  ((rank :reader array-marker-rank :initarg :rank)
   (initial-contents :reader array-marker-initial-contents :initarg :initial-contents)))
(defmethod marker-boundp ((marker array-marker)) t)
(defmethod marker-value ((marker array-marker))
  (loop with contents = (tangle (array-marker-initial-contents marker))
        repeat (array-marker-rank marker)
        for seq = contents then (and seq (elt seq 0))
        collect (length seq) into dimensions
        finally (return (make-array dimensions :initial-contents contents))))
(defun array-reader (stream sub-char arg)
  (declare (ignore sub-char))
  (unless arg (simple-reader-error stream "no␣rank␣supplied␣with␣#A"))
  (make-instance 'array-marker :rank arg :initial-contents (read stream t nil t)))
(dolist (mode '(:lisp :inner-lisp))
  (set-dispatch-macro-character #\# #\A #'array-reader (readtable-for-mode mode)))
```

107. Sharpsign S requires determining the standard constructor function of the structure type named, which we simply can't do portably. So 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 ((*readtable* (readtable-for-mode nil)))
    (values (read-from-string
             (write-to-string marker
                              :pprint-dispatch *tangle-pprint-dispatch*
                              :pretty t
                              :readably t)))))
(set-tangle-dispatch 'structure-marker
  (λ (stream obj)
    (format stream "#S~W" (structure-marker-form obj)))
  1)
(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)))
```

108. Sharpsign P reads namestrings and turns them into pathnames. We downcase logical pathname namestrings during tangling to avoid problems reading the output of implementations that conform to section 19.3.1.1.7 of the ANSI standard (e.g., SBCL) in ones that do not (e.g., Allegro and CCL).

```
(defclass pathname-marker (marker)
  ((namestring :reader pathname-marker-namestring :initarg :namestring)))
(defmethod marker-boundp ((marker pathname-marker)) t)
(defmethod marker-value ((marker pathname-marker))
  (parse-namestring (pathname-marker-namestring marker)))
(set-tangle-dispatch 'logical-pathname
  (λ (stream obj)
    (format stream "#P~W" (string-downcase (with-standard-io-syntax (namestring obj)))))
  1)
(set-tangle-dispatch 'pathname-marker
  (λ (stream obj)
    (format stream "#P~W" (etypecase (marker-value obj)
                            (logical-pathname (string-downcase (pathname-marker-namestring obj)))
                            (pathname (pathname-marker-namestring obj)))))
  1)
(defun pathname-reader (stream sub-char arg)
  (declare (ignore sub-char arg))
  (make-instance 'pathname-marker :namestring (read stream t nil t)))
(dolist (mode '(:lisp :inner-lisp))
  (set-dispatch-macro-character #\# #\P #'pathname-reader (readtable-for-mode mode)))
```

109. 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
    (symbol (not (null (member x *features* :test #'eq))))
    (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))))))
```

110. 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)))
(set-tangle-dispatch 'read-time-conditional
  (λ (stream obj)
    (format stream "#~:[-~;+~]~S~A"
            (read-time-conditional-plusp obj)
            (read-time-conditional-test obj)
            (read-time-conditional-form obj))))
(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-preserving-whitespace stream t nil nil)))
         (∗read-suppress∗ (if plusp (not (featurep test)) (featurep test))))
    (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)))
```

111. We do not preserve sharpsign vertical-bar comments during either weaving or tangling. We can therefore almost use the standard reader macro function, but it must be wrapped to obey our *read-maybe-nothing-internal* protocol.

```
(dolist (mode '(:lisp :inner-lisp))
  (set-dispatch-macro-character #\# #\|
                                (wrap-reader-macro-function (get-dispatch-macro-character #\# #\| nil))
                                (readtable-for-mode mode)))
```

112. Web control codes. 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))))
```

113. 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)))
```

114. 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 (readtable-for-mode :inner-lisp))
```

115. 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))))
```

116. 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 "@")))
```

117. The control code @q introduces is a CLWEB-only comment that will not be represented in either the tangled or woven results.

```
(set-control-code #\q (λ (stream sub-char arg)
                        (declare (ignore sub-char arg))
                        (read-line stream)
                        (values)))
```

118. While reading a `WEB` from a file, each section gets initialized with a *source-location* instance denoting the filename and line number at which the section began. (As for getting the pathname associated with the stream, only a best-effort is possible in general. It is reasonable to expect that any stream created by *tangle-file*, *weave*, and *load-web* will be covered by the below; only arbitrary streams supplied to *load-web* ought to cause problems. The *ignore-errors* for the *file-stream* case is meant to compensate for a defect in SBCL, whose definition of *file-stream* includes streams not created by *open*, on which *pathname* errors.)

```
(defstruct (source-location (:constructor create-source-location (lineno file)))
  lineno file)

(defun source-location-for-reader (stream)
  (let ((source-pathname (cond ((boundp '*weave-pathname*)
                                *weave-pathname*)
                               ((boundp '*tangle-file-pathname*)
                                *tangle-file-pathname*))))
    (when source-pathname
      (let ((lineno (stream-lineno stream)))
        (when lineno
          (create-source-location lineno source-pathname))))))
```

119. Ordinary sections are introduced by `@␣` or `@#\Newline`.

```
(defun start-section-reader (stream sub-char arg)
  (declare (ignore sub-char arg))
  (make-instance 'section :source-location (source-location-for-reader stream)))

(dolist (sub-char '(#\␣ #\Newline))
  (set-control-code sub-char #'start-section-reader '(:limbo :tex :lisp)))
```

120. Starred sections, introduced by `@*`, accept an optional numeric argument for the depth. Rather than the `CWEB` syntax in which the argument follows the '`*`', we follow the usual Common Lisp syntax and assume the argument occurs between the '`@`' and the '`*`'.

```
(defun start-starred-section-reader (stream sub-char arg)
  (declare (ignore sub-char))
  (apply #'make-instance 'starred-section
         :source-location (source-location-for-reader stream)
         (when arg `(:depth ,arg))))

(set-control-code #\* #'start-starred-section-reader '(:limbo :tex :lisp))
```

121. 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))
  (prog1 (if (and (char= (peek-char t stream t nil t) #\*)
                  (read-char stream t nil t))
             (apply #'make-instance 'starred-test-section :source-location (source-location-for-reader stream)
                    (when arg `(:depth ,arg)))
             (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))
```

122. 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)))
```

123. 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)
```

124. 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)))))
```

125. 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 127 ⟩)
          (if definition-allowed-p
              ⟨ Signal an error about section name use in TEX mode 128 ⟩
              (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)
```

126. ⟨ Condition classes 36 ⟩ +≡

```
(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)))))
```

127. ⟨ Signal an error about section definition in Lisp mode 127 ⟩ ≡

```
(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 125.

128. ⟨ Signal an error about section name use in TEX mode 128 ⟩ ≡

```
(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 125.

129. The control code `@w` reads the next two forms, which should be a symbol and a string, respectively. It tells the weaver to output the string instead of the symbol in Lisp mode.

```
(defun symbol-replacement-reader (stream sub-char arg)
  (declare (ignore sub-char arg))
  (let* ((symbol (read stream))
         (replacement (read stream)))
    (check-type symbol symbol "a␣symbol")
    (check-type replacement string "a␣replacement␣string")
    (weave-symbol-replace symbol replacement))
  (values))
(set-control-code #\w #'symbol-replacement-reader :lisp)
```

130. 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.

This control code probably won't be needed terribly often, since the indexer automatically adds the current package (i.e., the value of *package* at the time of indexing) to the list of packages whose symbols should be indexed.

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)
```

131. 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 (wrap-reader-macro-function #'index-entry-reader)
                    '(:tex :lisp)))
```

132. Reading sections. 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 139 ⟩
             (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 133 ⟩
       commentary
        ⟨ Finish the last section and initialize section variables 134 ⟩
        ⟨ Accumulate TEX-mode material in commentary 135 ⟩
       lisp
        ⟨ Accumulate Lisp-mode material in code 136 ⟩
       eof
        (push (finish-section section commentary code) sections)
        (return (nreverse sections))))))
```

133. 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 133 ⟩ ≡
(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 132.

134. ⟨ Finish the last section and initialize section variables 134 ⟩ ≡

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

This code is used in sections 132 and 137.

135. 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 135 ⟩ ≡
(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 132.

136. 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 136 ⟩ ≡
(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 137 ⟩)
          (newline-marker (when code (push form code)))
          (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 132.

This code is cited in section 139.

137.

```
⟨ Complain about starting a section without a commentary part 137 ⟩ ≡
(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 134 ⟩
```

This code is used in section 136.

138. ⟨ Condition classes 36 ⟩ +≡

```
(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))))))
```

139. 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* 136 ⟩.

⟨ Trim whitespace and reverse *commentary* and *code* 139 ⟩ ≡

```
(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 132.

140. Tangling. 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))
      (named-section
       (if expand-named-sections
           (values (section-code form) t)
           (values form nil)))
      (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)))))))))
```

141. *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)))
```

142. 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)))
```

143. 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 25 ⟩
  (when verbose (format t "~&;␣loading␣WEB␣from␣~S~%" filespec))
  (if (streamp filespec)
      (load-web-from-stream filespec print)
      (with-open-file (stream (input-file-pathname filespec)
                       :direction :input
                       :external-format external-format
                       :if-does-not-exist (if if-does-not-exist :error nil))
        (load-web-from-stream stream print))))
```

144. 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 (file (probe-file file)))
  "Load␣web␣sections␣from␣FILE,␣then␣delete␣it.␣If␣APPEND␣is␣true,␣named
 section␣definitions␣in␣FILE␣will␣be␣appended␣to␣existing␣definitions;
 otherwise,␣they␣will␣replace␣them."
  (when file
    (unwind-protect
         (with-open-file (stream file :direction :input)
           (load-web-from-stream stream t :append append))
      (delete-file file))))
```

145. Some tests might not be worth spending time compiling, or might be easier to debug if not compiled, or might involve unwanted complexity to support compilation (e.g., *make-load-form*). So we'll allow the user to specify whether to compile a tests file, with a variable for the default preference.

```
⟨ Global variables 12 ⟩ +≡
(defvar ∗compile-tests-file∗ t
  "The␣default␣value␣for␣the␣:COMPILE-TESTS-FILE␣argument␣to␣TANGLE-FILE.")
```

146. During tangling, we bind ∗*tangle-file-pathname*∗ and ∗*tangle-file-truename*∗ in the same way that *compile-file* binds ∗*compile-file-pathname*∗ and ∗*compile-file-truename*∗.

⟨ Global variables 12 ⟩ +≡
```
(defvar ∗tangle-file-pathname∗ nil)
(defvar ∗tangle-file-truename∗ nil)
```

147. We'll use a custom pprint-dispatch table for tangling to avoid cluttering the default table.

⟨ Global variables 12 ⟩ +≡
```
(defparameter ∗tangle-pprint-dispatch∗ (copy-pprint-dispatch nil))
```

148. ⟨ Define the pprint dispatch table setters 148 ⟩ ≡
```
(defun set-tangle-dispatch (type-specifier function &optional (priority 0))
  (set-pprint-dispatch type-specifier function priority ∗tangle-pprint-dispatch∗))
```
See also section 165.

This code is used in section 74.

149. ANSI Common Lisp's *open* specification was intended to allow an implementation to supply semantics for the various *:if-exists* modes consistent with the conventions of the file system with which the program interacted, including, notably, allowing the default *:if-exists* disposition of *:new-version* to have the same semantics as some other opening mode. The implementors of SBCL, however, chose to needlessly inconvenience their users by making the default *:if-exists* disposition signal an error.

⟨ *:if-exists* default disposition 149 ⟩ ≡
```
#+:sbcl :supersede
#−:sbcl :new-version
```
This code is used in sections 150 and 161.

150. 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. We compile the tests file after the source file on the assumption that the tests probably depend on the main program, but we return the values returned by the call to *compile-file* on the tangled file so that you can say, e.g., (*load* (*tangle-file* "web")).

```
(defun tangle-file (input-file &rest args &key output-file tests-file
                    (verbose *compile-verbose*)
                    (print *compile-print*)
                    (if-exists ⟨ :if-exists default disposition 149 ⟩)
                    (compile-file t)
                    (compile-tests-file *compile-tests-file*)
                    (external-format :default) &allow-other-keys &aux
                    (input-file (input-file-pathname input-file))
                    (readtable *readtable*)
                    (package *package*))
  "Tangle and compile the web in INPUT-FILE, producing OUTPUT-FILE."
  (declare (ignorable output-file tests-file))
  (multiple-value-bind (output-file lisp-file tests-output-file tests-file) (apply #'tangle-file-pathnames input-file args)
    (with-standard-io-syntax
      (let* ((*readtable* readtable)
             (*package* package)
             (*print-readably* nil)
             (*tangle-file-pathname* input-file)
             (*tangle-file-truename* (truename *tangle-file-pathname*)))
        (when verbose (format t "~&; tangling web from ~A:~%" input-file))
        ⟨ Initialize global variables 25 ⟩
        (with-open-file (input input-file :direction :input :external-format external-format)
          (read-sections input))
        ⟨ Complain about any unused named sections 152 ⟩
        ⟨ Ensure that every referenced named section is defined 154 ⟩
        (cond ((and tests-file (> (length *test-sections*) 1))
               (when verbose (format t "~&; writing tests to ~A~%" tests-file))
               (tangle-sections *test-sections*
                                :input-file input-file :output-file tests-file
                                :if-exists if-exists :external-format external-format))
              (t (setq tests-file nil)))
        (when verbose (format t "~&; writing tangled code to ~A~%" lisp-file))
        (tangle-sections *sections*
                         :input-file input-file :output-file lisp-file
                         :if-exists if-exists :external-format external-format)
        (if compile-file
            (with-compilation-unit ⟨ Compilation unit for tangling 151 ⟩
              (multiple-value-prog1
                  (compile-file lisp-file
                                :output-file output-file
                                :verbose verbose :print print
                                :external-format external-format)
                (when (and tests-file compile-tests-file)
                  (compile-file tests-file
                                :output-file tests-output-file
                                :verbose verbose :print print
```

```
                                :external-format external-format))))
        lisp-file)))))
```

151. The function *make-definition-location-table* can build a fasdumpable data structure associating symbols with the line numbers of their definitions. Here we arrange for SBCL to dump this data structure into FASLs during compilation of a tangled web.

⟨ Compilation unit for tangling 151 ⟩ ≡

```
#+:sbcl
(:source-namestring (namestring *tangle-file-pathname*)
 :source-plist (list :clweb-section-linenos
                     (make-definition-location-table
                      (remove-if #'test-section-p *sections*))))
#-:sbcl
()
```

This code is used in section 150.

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

⟨ Complain about any unused named sections 152 ⟩ ≡

```
(let ((unused-sections '()))
  (flet ((note-unused-section (section)
           (when (null (used-by section))
             (push section unused-sections))))
    (map-bst #'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 150.

153. ⟨ Condition classes 36 ⟩ +≡

```
(define-condition unused-named-section-warning (simple-warning) ())
```

154. Named sections that are referenced but never defined are assumed to be errors. Here we'll rely on the fact that *section-number* detects this condition and signals an error if it occurs.

⟨ Ensure that every referenced named section is defined 154 ⟩ ≡

```
(map-bst #'section-number *named-sections*)
```

This code is used in section 150.

155. *tangle-file* uses this helper function to actually write the tangled forms to the intermediate Lisp source file. It's used for both ordinary and test sections.

```
(defun tangle-sections (sections &key input-file output-file if-exists external-format)
  (with-open-file (output output-file
                          :direction :output
                          :if-exists if-exists
                          :external-format external-format)
    (format output ";;;;␣TANGLED␣WEB␣FROM␣\"~A\".␣DO␣NOT␣EDIT.~%"
            (enough-namestring input-file))
    (let ((*evaluating* nil)
          (*print-pprint-dispatch* *tangle-pprint-dispatch*)
          (*print-readably* nil))
      ⟨ Output a form that sets the source pathname 156 ⟩
      (dolist (form (tangle (unnamed-section-code-parts sections)))
        (pprint form output)))))
```

156. Allegro Common Lisp uses the variable **source-pathname** to locate definitions in source files. If we override that, we can get it to look in the `CLWEB` file instead of the tangled Lisp file.

⟨ Output a form that sets the source pathname 156 ⟩ ≡

```
#+:allegro
(format output
        "#+ALLEGRO~
         ~&(EVAL-WHEN (:COMPILE-TOPLEVEL :LOAD-TOPLEVEL)~
         ~&  (SETQ EXCL:*SOURCE-PATHNAME* ~S))~%"
        (namestring *tangle-file-pathname*))
```

This code is used in section 155.

157. Weaving. The function *weave* reads a web from *input-file* and produces an output TEX file named by *output-file*. By default, it also weaves test sections to the file named by *tests-file* and produces an index and list of section names which can be included from the output file. The defaulting behavior for the filename arguments is complex; see below.

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 a '∗'. 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
              (verbose ∗weave-verbose∗)
              (print ∗weave-print∗)
              ((:source-locations ∗weave-source-locations∗)
               ∗weave-source-locations∗)
              (external-format :default) &aux
              (input-file (input-file-pathname input-file))
              (readtable ∗readtable∗)
              (package ∗package∗)
              ∗weave-pathname∗)
  "Weave␣the␣web␣contained␣in␣INPUT-FILE,␣producing␣the␣TeX␣file␣OUTPUT-FILE."
  (declare (ignorable output-file tests-file index-file))
  (multiple-value-bind (output-file index-file sections-file) (apply #'weave-pathnames input-file args)
    (setq ∗weave-pathname∗ input-file)
    (with-standard-io-syntax
      (let ((∗readtable∗ readtable)
            (∗package∗ package)
            (∗print-readably∗ nil))
        (when verbose (format t "~&;␣weaving␣web␣from␣~A:~%" input-file))
        ⟨ Initialize global variables 25 ⟩
        (with-open-file (input input-file :direction :input :external-format external-format)
          (read-sections input))
        (let ((tests-file (apply #'tests-file-pathname input-file :output-file output-file args)))
          (when (and tests-file (> (length ∗test-sections∗) 1))
            (when verbose (format t "~&;␣weaving␣tests␣to␣~A~%" tests-file))
            (multiple-value-bind (output-file index-file sections-file)
                (apply #'weave-pathnames input-file :output-file tests-file args)
              ⟨ Fix up the index and sections filenames for test suite output 159 ⟩
              (weave-sections ∗test-sections∗
                              :input-file input-file :output-file output-file
                              :index-file index-file :sections-file sections-file
                              :verbose verbose :print print
                              :external-format external-format
                              :weaving-tests t))))
        (when verbose (format t "~&;␣weaving␣sections␣to␣~A~%" output-file))
        (weave-sections ∗sections∗
                        :input-file input-file :output-file output-file
                        :index-file index-file :sections-file sections-file
                        :verbose verbose :print print
                        :external-format external-format)))))
```

158. The weaver also binds a special variable, mostly for source location tracking.

⟨ Global variables 12 ⟩ +≡
```
(defvar *weave-pathname*)
```

159. We don't accept a separate argument for the test suite index filename, so we have to do a little post-processing to get it all right. At this point, *output-file* is the name of the *tests* output file; we'll use its name component as the name components of the index and sections files for the test suite.

⟨ Fix up the index and sections filenames for test suite output 159 ⟩ ≡
```
(when index-file
  (let ((output-file-name (pathname-name output-file)))
    (setq index-file (make-pathname :name output-file-name :defaults index-file)
          sections-file (make-pathname :name output-file-name :defaults sections-file))))
```
This code is used in section 157.

160. ⟨ Global variables 12 ⟩ +≡
```
(defvar *weave-verbose* t
  "The␣default␣for␣the␣:VERBOSE␣argument␣to␣WEAVE.")
(defvar *weave-print* t
  "The␣default␣for␣the␣:PRINT␣argument␣to␣WEAVE.")
(defvar *weave-source-locations* t
  "If␣true,␣include␣src:␣specials␣in␣the␣woven␣TeX.")
```

161. The following routine does most of the actual output for the weaver. It takes a list of sections and the defaulted arguments from *weave*, and writes the output file and possibly the index and section list files as well. It's used to produce both the primary and test output, which differ only slightly; the *weaving-tests* argument will be true if it's currently weaving the tests.

```
(defun weave-sections (sections &key input-file output-file index-file sections-file
                       weaving-tests verbose print
                       (if-exists ⟨ :if-exists default disposition 149 ⟩)
                       external-format)
  (macrolet ((with-output-file ((stream filespec) &body body)
               `(with-open-file (,stream ,filespec
                                 :direction :output
                                 :external-format external-format
                                 :if-exists if-exists)
                  ,@body)))
    (with-output-file (out output-file)
      (format out "\\input␣clwebmac~%")
      (when weaving-tests ⟨ Write the program name to the tests output file 163 ⟩)
      (if print
          ⟨ Weave the sections to the output file, reporting as we go 162 ⟩
          (map nil (λ (section) (weave-object section out)) sections))
      (when index-file
        (when verbose (format t "~&;␣writing␣the␣index␣to␣~A~%" index-file))
        (with-output-file (idx index-file)
          (weave-object (index-sections sections :index (if weaving-tests (make-index) ∗index∗))
                        idx))
        (with-output-file (scn sections-file)
          (map-bst (λ (section)
                     (unless (every (if weaving-tests (complement #'test-section-p) #'test-section-p)
                                    (named-section-sections section))
                       (weave-object (make-section-name-index-entry section) scn)))
                   ∗named-sections∗))
        (format out "~&\\inx~%\\fin~%\\con~%"))
      (format out "~&\\end~%")
      (truename out))))
```

162. We'll use the pretty printer to print the section numbers to standard output as we weave them. Starred sections get a '`*`' prefix, and the whole of the output is in the form of a comment.

⟨ Weave the sections to the output file, reporting as we go 162 ⟩ ≡

```
(flet ((weave-section (section)
         (format t "~:[~;*~]~D" (starred-section-p section) (section-number section))
         (weave-object section out)))
  (pprint-logical-block (nil (coerce sections 'list) :per-line-prefix ";␣␣")
    (weave-section (pprint-pop))
    (loop
      (pprint-exit-if-list-exhausted)
      (write-char #\␣)
      (pprint-newline :fill)
      (weave-section (pprint-pop)))))
```

This code is used in section 161.

163. In the tests output file, we generate references to the sections of the main program that are presumably being tested. To produce those references, the TEX macros `\T` and `\Ts` use the `\progname` macro. We provide a default definition here using the name of the input file, but the user can easily override this by giving an alternate definition in the limbo text.

⟨ Write the program name to the tests output file 163 ⟩ ≡

```
(format out "\\def\\progname{~/clweb::print-escaped/}~%"
        (if input-file
            (string-capitalize (pathname-name input-file))
            "program"))
```

This code is used in section 161.

164. Printing the woven output. 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))
```

165. ⟨ Define the pprint dispatch table setters 148 ⟩ +≡

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

166. Here's a little convenience routine for weaving an object to a stream.

```
(defun weave-object (object stream)
  (write object
         :stream stream
         :case :downcase
         :pretty t :pprint-dispatch *weave-pprint-dispatch* :right-margin 1000))
```

167. 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))))))
```

168. 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))))
```

169. Printing a limbo section is simple: we just dump out the TEX text. Notice that we use a priority of 1 so as to override the normal section object printer, which we'll come to next.

```
(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)
```

170. Section objects are printed just like any other objects, but they use some special TEX macros to set up the formatting. They may be preceded by `\SL`, which sets up source location information for some DVI previewers; but always commence with either `\M` or `\N` depending on whether the section is unstarred or starred. Then comes the commentary, which is separated from the code part by a bit of vertical space using the `\Y` macro if both are present. The code part always starts with `\B`, followed by the name if it's a named section. Then comes the code, which we output one form at a time, using alignment tabs (`\+`). Last come the cross-references and a final `\fi` that matches the `\ifon` in `\M` and `\N`.

```
(defun print-section (stream section)
  (format stream "~&~:[~*~;~@[\\SL~W~]~%~]\\~:[M~*~;N{~D}~]{~D}"
          *weave-source-locations*
          (section-source-location section)
          (starred-section-p section)
          (section-depth section)
          (section-number section))
  (let* ((commentary (section-commentary section))
         (name (section-name section))
         (named-section (and name (find-section name)))
         (code (section-code section))
         (test-section-p (typep section 'test-section)))
    (print-tex stream commentary)
    (cond ((and commentary code) (format stream "~&\\Y\\B~%"))
          (code (format stream "~&\\B~%")))
    (when named-section
      (print-section-name stream named-section)
      (format stream "${}~:[\\mathrel+~;~]\\E{}$\\6~%"
              (= (section-number section) (section-number named-section))))
    (when code
      (dolist (form code)
        (typecase form
          (newline-marker (format stream "~W" form))
          (t (format stream "~@<\\+~@;~W~;\\cr~:>" form))))
      (format stream "~&\\egroup~%")) ; matches \bgroup in \B
    (when (and (not named-section)
               (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)))
      (flet ((filter-xref (xref-section)
               (or (eql xref-section section)
                   (if test-section-p
                       nil
                       (typep xref-section 'test-section)))))
        (print-xrefs stream #\A (remove-if #'filter-xref (named-section-sections named-section)))
        (print-xrefs stream #\U (remove-if #'filter-xref (used-by named-section)))
        (print-xrefs stream #\Q (remove-if #'filter-xref (cited-by named-section))))))
  (format stream "~&\\fi~%"))
(set-weave-dispatch 'section #'print-section)
```

171. Source locations are printed in a bespoke syntax implemented by the `\SL` macro.

```
(defun print-source-location (stream source-location)
  (format stream "[~D␣~A]"
          (source-location-lineno source-location)
          (file-namestring (translate-logical-pathname
                            (source-location-file source-location)))))
(set-weave-dispatch 'source-location #'print-source-location)
```

172. 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) #'<))))
```

173. Aside from printing the section name in the body of the woven output, this next 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)
```

174. 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)))
(defun make-section-name-index-entry (section)
  (make-instance 'section-name-index-entry :named-section 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)))))
```

175. 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* '(("␣\\%&#$^_" . #\\)
                                    ("~" . "$\\sim$")
                                    ("{" . "$\\{$") ("}" . "$\\}$")
                                    ("<" . "$<$") (">" . "$>$")))
(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)))
```

176. We print strings one line at a time, being careful to properly end each alignment row. As a special nicety for *format* strings, if the last character before a newline is a '~', we skip any whitespace following the newline.

```
(defun print-string (stream string)
  (loop with *print-escape-list* = `(("{*~}" . #\\)
                                     ("\\" . "\\\\\\\\")
                                     ("\"" . "\\\\\"")
                                     ,@*print-escape-list*)
        for last = 0 then (if (char= (elt string (1- newline)) #\~)
                              (position-if-not #'whitespacep string :start newline)
                              (1+ newline))
        for newline = (position #\Newline string :start last)
        as line = (subseq string last newline)
        do (format stream "~:[~;\\.{~]~:[~;\"~]~/clweb::print-escaped/~:[~;\"~]~:[~;}~]"
                   *print-escape* (and *print-escape* (zerop last))
                   line (and *print-escape* (null newline)) *print-escape*)
        while newline do (format stream "\\cr~:@_")))
(set-weave-dispatch 'string #'print-string)
```

177. We print non-graphic characters using their names, when available.

```
(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
                (string char)))
    char))
(set-weave-dispatch 'character #'print-char)
```

178. We transform `#'` into instances of *function-marker* during reading, so we don't want lists of the form (*function foo*) turned into *#'foo* during weaving.

```
(set-weave-dispatch '(cons (eql function))
  (λ (stream obj)
    (format stream "(~{~W~^ ~})" obj))
  1)
```

179. Symbols receive special treatment during weaving. Lambda-list keywords and symbols in the keyword package have specialized TEX macros that add a bit of emphasis.

```
(defun print-symbol (stream symbol)
  (flet ((print-case (string)
           (funcall (ecase *print-case*
                      (:upcase #'string-upcase)
                      (:downcase #'string-downcase)
                      (:capitalize #'string-capitalize))
                    string)))
    (let* ((group-p (cond ((member symbol lambda-list-keywords) (write-string "\\K{" stream))
                          ((keywordp symbol) (write-string "\\:{" stream))))
           (symbol-name (symbol-name symbol))
           (package (symbol-package symbol)))
      ⟨ Maybe print a package prefix for symbol 180 ⟩
      ⟨ Print symbol-name 181 ⟩
      (when group-p (write-string "}" stream)))))
(set-weave-dispatch 'symbol #'print-symbol)
```

180. ⟨ Maybe print a package prefix for *symbol* 180 ⟩ ≡

```
(cond ((keywordp symbol)) ; TEX macro supplies the colon
      ((eq package *package*)) ; no prefix necessary
      ((null package) (write-string "\\#:" stream))
      (t (multiple-value-bind (found accessible) (find-symbol symbol-name *package*)
           (unless (and accessible (eq found symbol))
             (print-escaped stream (print-case (package-name package)))
             (case (nth-value 1 (find-symbol symbol-name package))
               (:external (write-char #\: stream))
               (otherwise (write-string "::" stream)))))))
```

This code is used in section 179.

181. ⟨ Print *symbol-name* 181 ⟩ ≡

```
(multiple-value-bind (prefix suffix) ⟨ Split symbol-name into a prefix and nicely formatted suffix 182 ⟩
  (print-escaped stream (print-case prefix))
  (when suffix (write-string suffix stream)))
```

This code is used in section 179.

182. Symbols with certain suffixes get a bit of fancy formatting. We test for suffixes one at a time, and if we find a match, we return two values: the portion of the symbol name before the suffix, and the suffix replacement. Otherwise, we just return the symbol name.

⟨ Split *symbol-name* into a prefix and nicely formatted suffix 182 ⟩ ≡

```
(loop with length = (length symbol-name)
      for (suffix . replacement) in *print-symbol-suffixes*
      as prefix-end = (max 0 (- length (length suffix)))
      when (string= symbol-name suffix :start1 prefix-end)
       do (return (values (subseq symbol-name 0 prefix-end) replacement))
      finally (return symbol-name))
```

This code is used in section 181.

183. We keep the suffixes in a global association list in case the user wants to override or augment them. The entries are pairs of strings of the form '(⟨*suffix*⟩ . ⟨*replacement*⟩)'.

⟨ Global variables 12 ⟩ +≡

```
(defvar *print-symbol-suffixes*
  '(("/=" . "$\\neq$")
    ("<=" . "$\\leq$")
    (">=" . "$\\geq$")
    ("+-" . "$\\pm$")
    ("+" . "$+$")
    ("-" . "$-$")
    ("=" . "$=$")
    ("'" . "$'$")))
```

184. A few symbols get special replacements by default, e.g., λ and π. Additional replacements may be added via the `@w` control code, which calls this function.

```
(defun weave-symbol-replace (symbol replacement)
  (set-weave-dispatch `(eql ,symbol)
    (λ (stream obj)
      (declare (ignore obj))
      (write-string replacement stream))
    1))
(weave-symbol-replace 'λ "\\L")
(weave-symbol-replace 'π "$\\pi$")
```

185. Indentation tracking. 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.

```
(defclass logical-block ()
  ((list :reader logical-block-list :initarg :list)))
(defun make-logical-block (list)
  (make-instance 'logical-block :list list))
```

186. 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 187 ⟩))
    ;; 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)))))
```

187. 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 can make the resulting TEX somewhat simpler.

⟨ Build a logical block from *list* 187 ⟩ ≡

```
(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 (< 0 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 186.

188. 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)
```

189. ...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~:[~;\\Y~]~:@_" (typep obj 'par-marker))
             (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)
```

190. Printing markers. 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.

191.

```
(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)
```

192.

```
(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)))
```

193.

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

194.

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

195.

```
(set-weave-dispatch 'backquote-form
  (λ (stream obj)
    (format stream "\\`~W" (cadr obj))))
(set-weave-dispatch 'comma
  (λ (stream obj)
    (format stream "\\CO{~@[~C~]}~W"
            (comma-modifier obj)
            (comma-form obj :tangle nil))))
```

196.
```
(set-weave-dispatch 'function-marker
  (λ (stream obj)
    (format stream "\\#\\'~S" (quoted-form obj)))
  1)
```

197.
```
(set-weave-dispatch 'simple-vector-marker
  (λ (stream obj)
    (format stream "\\#~@[~D~]~W"
            (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)
```

198.
```
(set-weave-dispatch 'read-time-eval-marker
  (λ (stream obj)
    (format stream "\\#.~W" (read-time-eval-form obj))))
```

199.
```
(set-weave-dispatch 'radix-marker
  (λ (stream obj)
    (format stream "$~VR_{~2:*~D}$"
            (radix-marker-base obj) (marker-value obj))))
```

200.
```
(set-weave-dispatch 'complex-marker
  (λ (stream obj)
    (format stream "\\#C\\/~W" (complex-components obj))))
```

201.
```
(set-weave-dispatch 'array-marker
  (λ (stream obj)
    (format stream "\\#~D\\/A~W"
            (array-marker-rank obj)
            (array-marker-initial-contents obj))))
```

202.
```
(set-weave-dispatch 'structure-marker
  (λ (stream obj)
    (format stream "\\#S\\/~W" (structure-marker-form obj))))
```

203.

```
(set-weave-dispatch 'pathname-marker
  (λ (stream obj)
    (format stream "\\#P\\/~W" (pathname-marker-namestring obj))))
```

204. Read-time conditionals, like literal strings, must be printed one line at a time so as to respect the alignment tabs. Since the `\RC` macro switches to a monospaced font and activates `\obeyspaces`, dealing with indentation is straightforward.

```
(set-weave-dispatch 'read-time-conditional-marker
  (λ (stream obj)
    (let* ((form (read-time-conditional-form obj))
           (lines ⟨ Split form into lines, eliding any initial indentation 205 ⟩)
           (*print-escape-list* `(("␣" . "␣") ,@*print-escape-list*)))
      (flet ((print-rc-prefix ()
               (format stream "\\#$~:[-~;+~]$~S"
                       (read-time-conditional-plusp obj)
                       (read-time-conditional-test obj)))
             (print-rc-line (line)
               (format stream "\\RC{~/clweb::print-escaped/}" line)))
        (cond ((= (length lines) 1)
               (print-rc-prefix)
               (print-rc-line (first lines)))
              (t (write-string "\\!" stream)
                 (pprint-logical-block (stream lines :per-line-prefix "&")
                   (print-rc-prefix)
                   (pprint-exit-if-list-exhausted)
                   (loop for line = (pprint-pop)
                         when (plusp (length line)) do (print-rc-line line)
                         do (pprint-exit-if-list-exhausted)
                            (format stream "\\cr~:@_")))))))))
```

205. ⟨ Split *form* into lines, eliding any initial indentation 205 ⟩ ≡

```
(loop with indent = (if (char= (elt form 0) #\Newline)
                        (1- (position-if-not #'whitespacep form :start 1))
                        0)
      for last = 0 then (+ indent newline 1)
      for newline = (position #\Newline form :start last)
      as line = (subseq form last newline)
      collect line while newline)
```

This code is used in section 204.

206. Code walking. 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.

207. 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.

```
(defun ensure-portable-walking-environment (env)
  #+:allegro (sys:ensure-portable-walking-environment env)
  #-:allegro env)
```

208. Allegro doesn't define *parse-macro* or *enclose*, so we'll need to provide our own definitions.

We'll use our own version of *enclose* on all implementations. Thanks to Duane Rettig of Franz, Inc. for the idea behind this implementation (post to *comp.lang.lisp*, message-id ⟨*4is97u4vv.fsf@franz.com*⟩, 2004-10-18).

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

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

```
#+:allegro
(defun parse-macro (name lambda-list body &optional env)
  (declare (ignorable name lambda-list body env))
  (excl::defmacro-expander `(,name ,lambda-list ,@body) env))
```

210. 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)
```

211. Along with the environment, most of the walker functions take a 'context' argument that gives some extra information about the form being walked. Those arguments will always be instances of *walk-context*, or *nil* if there is no relevant information to supply.

```
(defclass walk-context () ())
(defun make-context (context &rest args)
  (apply #'make-instance context args))
```

212. The most important kind of context we'll supply is the *namespace* of a name that's about to be walked. Namespaces are associated with environments, but the CLtL-2 environments API does not provide functions for dealing directly with many of the kinds of namespaces that we'll define. However, the most important namespaces—variable names, function names, macros, *&c.*—do have corresponding entries in the lexical environment objects that we'll pass around, and we'll make it easy to maintain this link.

The main problem that these namespace objects solve is that we'll need to walk names of all kinds *before* we add them to or even look them up in the environment because of the referring symbols the indexing walker uses to track sections in tangled code. We'll therefore use them to represent both a namespace that an evaluated name should already exist in, and the namespace in which a new binding is being established.

Associated with every namespace is a name—not the name that's being walked, but an identifier for the namespace itself, like 'variable' or 'function'. Those names will be keyword symbols, and for namespaces that correspond directly to lexical environments, the name should match the symbol returned as the second value from the environment accessor for that namespace. For instance, *variable-information* returns *:lexical* as its second value if the given variable is a lexical variable in the current environment, and so our lexical variable namespace has *:lexical* as its name.

Every entry in a namespace—i.e., every binding—can be either 'local' or 'global'. For some namespaces, only global names are supported (e.g., compiler macros can only be global), but the question "is this binding local or global?" still makes sense for those namespaces.

```
(defclass namespace (walk-context)
  ((name :reader namespace-name :initform nil :allocation :class)
   (local :reader local-binding-p :initarg :local :type boolean))
  (:default-initargs :local nil))
```

213. During a walk, we'll sometimes need to go from a namespace name to the class we use to represent that namespace. The following hash table and accessor functions implement that mapping.

```
(defparameter *namespace-classes* (make-hash-table :test 'eq))
(defun (setf find-namespace-class) (class namespace-name)
  (setf (gethash namespace-name *namespace-classes*) class))
(defun find-namespace-class (namespace-name)
  (flet ((read-namespace-class-name ()
           (loop
             (format *query-io* "Enter a namespace class name: ")
             (let ((class (find-class (read *query-io*) nil)))
               (when (and class (subtypep class 'namespace))
                 (return (list class)))))))
    (restart-case
        (or (gethash namespace-name *namespace-classes*)
            (error "Can't find namespace class for namespace ~S." namespace-name))
      (use-value (value)
        :report "Specify a class to use this time."
        :interactive read-namespace-class-name
        value)
      (store-value (value)
        :report "Specify a class to store and use in the future."
        :interactive read-namespace-class-name
        (setf (find-namespace-class namespace-name) value)))))
```

214. We'll wrap up the namespace class definitions in a little defining macro that eliminates some syntactic redundancies and sets up the namespace name → namespace class mapping.

```
(defmacro defnamespace (class-name (&rest supers) &optional namespace-name other-slot-specs)
  `(progn
     (defclass ,class-name ,(or supers '(namespace))
       (,@(when namespace-name
            `((name :initform ',namespace-name :allocation :class)))
        ,@other-slot-specs))
     ,@(when namespace-name
         `((setf (find-namespace-class ',namespace-name) (find-class ',class-name))))))
```

⟨ Define namespace classes 215 ⟩

215. And here are the basic namespace definitions themselves. We've put them in a named section so that we can add to the list later and not have to worry about forward references.

⟨ Define namespace classes 215 ⟩ ≡
```
(defnamespace variable-name () :variable)
(defnamespace lexical-variable-name (variable-name) :lexical)
(defnamespace special-variable-name (variable-name) :special)
(defnamespace constant-name () :constant)
(defnamespace symbol-macro-name () :symbol-macro)

(defnamespace operator () :operator)
(defnamespace function-name (operator) :function)
(defnamespace setf-function-name (function-name) :setf-function)
(defnamespace macro-name (operator) :macro)
(defnamespace compiler-macro-name (operator) :compiler-macro)
(defnamespace special-operator (operator) :special-form)

(defnamespace block-name () :block)
(defnamespace tag-name () :tag)
(defnamespace catch-tag () :catch-tag)
(defnamespace type-name () :type)
```
See also sections 216, 217, 347, 351, 355, 359, and 361.

This code is used in section 214.

216. *class-name* is a symbol in the `COMMON-LISP` package, so we can't use it for the name of a class.

⟨ Define namespace classes 215 ⟩ +≡
```
(defnamespace class-name% () :class)
(defnamespace condition-class-name (class-name%) :condition-class)
```

217. We'll use two additional namespaces for macros and symbol macros, because when we add those to the lexical environments, we'll need to supply the definitions as well. Using seperate namespace classes makes the bookkeeping slightly simpler.

⟨ Define namespace classes 215 ⟩ +≡
```
(defnamespace symbol-macro-definition (symbol-macro-name))
(defnamespace macro-definition (macro-name))
```

218. The people at Franz just can't seem to resist tweaking the environments API: their version of *function-information* returns *:sepcial-operator* rather than *:special-form* when the function spec names a special operator. This may be more precise terminology, but it's an arbitrary and capricious change. Still, it's easy enough to provide an alias for our namespace class.

```
(setf (find-namespace-class :special-operator) (find-class 'special-operator))
```

219. The reason the association between namespace names and environment types is important is that when we walk a name, we can sometimes only give a best guess as to the correct namespace. For instance, when we walk an evaluated symbol, it could be a lexical variable, a special variable, a constant, or a symbol macro—we can't know until we look it up in the environment. But in general, we *can't* look it up until we've walked it; the best we can do is say that it's a variable of some kind. Once the walk happens, we can get more information from the environment, and that's exactly what we'll do.

The function *update-context* takes a (walked) name, a context instance, and an environment object, and returns a new context that reflects the given context as updated by the appropriate environment entry for that name. It uses the mapping between namespace names and environment types, which we'll retrieve using *find-namespace-class*.

```
(defgeneric update-context (name context env))

(defmethod update-context (name (context walk-context) env)
  (declare (ignore name env))
  context)

(defmethod update-context (name (context variable-name) env)
  (multiple-value-bind (type local) (variable-information name env)
    (if type
        (make-context (find-namespace-class type) :local local)
        context)))
```

220. For function names, we need to check if the function is a generic function, then what type it has in the environment. If the name is being bound, it won't necessarily be in the environment yet, so we'll just return the given context.

However, just because we don't find the name in the environment doesn't mean it isn't there: neither Allegro nor CCL correctly support looking up a function name of the form '(*setf* ⟨*symbol*⟩)' in an arbitrary lexical environment, and SBCL versions prior to 1.0.47.30 signaled an error due to a type declaration bug (whence the careful error handling). We therefore special-case the non-atomic-function-name case, which really shouldn't be necessary.

```
(deftype setf-function () '(cons (eql setf) (cons symbol null)))

(defmethod update-context (name (context operator) env)
  (multiple-value-bind (type local) (handler-case (function-information name env)
                                      (type-error () (values nil nil)))
    (cond ((and (not local) (generic-function-p name))
           (make-context (etypecase name
                           (symbol 'generic-function-name)
                           (setf-function 'generic-setf-function-name))))
          ((or type (not (symbolp name)))
           (make-context (find-namespace-class
                          (etypecase name
                            (symbol type)
                            (setf-function :setf-function)))
                         :local (or local (local-binding-p context))))
          (t context))))
```

221. Macro definitions have to be handled separately, but they're much simpler than functions.

```
(defmethod update-context (name (context macro-definition) env)
  (multiple-value-bind (type local) (function-information (car name) env)
    (if type
        (make-context (find-namespace-class type) :local local)
        context)))
```

222. Now we come to the walker proper. Code walkers are represented as instances of the class *walker*. The walker protocol consists of a handful of generic functions, which we'll describe as we come to them; subclasses of *walker* may add methods to them to customize the behavior of the walk.

```
(defclass walker () ())
⟨ Walker generic functions 224 ⟩
```

223. We'll wrap *macroexpand-1* so that walker sub-classes get a chance to override it.

```
(defmethod macroexpand-for-walk ((walker walker) form env)
  (macroexpand-1 form env))
```

224. ⟨ Walker generic functions 224 ⟩ ≡

```
(defgeneric macroexpand-for-walk (walker form env))
```

See also sections 226, 230, 232, 237, 239, and 257.

This code is used in section 222.

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

```
(defmethod walk-form ((walker walker) form &optional env toplevel &aux
                      (env (ensure-portable-walking-environment env)))
  (let ((expanded t))
    (loop
      (setq form (catch 'continue-walk ⟨ Walk form 227 ⟩))
      (multiple-value-setq (form expanded)
        (macroexpand-for-walk walker form env)))))
```

226. ⟨ Walker generic functions 224 ⟩ +≡

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

227. When we hit an atom that is not a symbol macro, a compound form whose car is not a macro, or a special form, we return the walked form and break out of the macro-expansion loop.

⟨ Walk *form* 227 ⟩ ≡

```
(cond ((atom form)
       ⟨ Walk the atom form 228 ⟩)
      ((not (symbolp (car form)))
       (return (walk-list walker form env toplevel)))
      ((or (not expanded) (walk-as-special-form-p walker (car form) form env))
       (return (walk-compound-form walker (car form) form env :toplevel toplevel)))
      (t form))
```

This code is used in section 225.

228. When we walk an atomic form, we need to check whether or not it's a symbol macro. If it is, we'll let the walk continue with the expansion; otherwise, we'll just return the walked atom. But it's important to walk the atom *before* checking whether it's a symbol macro because of the games we play later with referring symbols in the indexer.

```
⟨ Walk the atom form 228 ⟩ ≡
(typecase form
  (symbol
   (let ((var (walk-name walker form (make-context 'variable-name) env)))
     (if (eql (variable-information var env) :symbol-macro)
         var ; wait for macro expansion
         (return var))))
  (t (return (walk-atomic-form walker form nil env :toplevel toplevel))))
```

This code is used in section 227.

229. The walker will treat a compound form as a special form if and only if *walk-as-special-form-p* returns true of that form. Besides the walker instance, the form being walked, and the current environment, we'll supply the operator of the compound form as a separate argument so that we can use *eql* specializers. The *operator* should generally be *eql* to (*car form*).

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

230.

```
⟨ Walker generic functions 224 ⟩ +≡
(defgeneric walk-as-special-form-p (walker operator form env))
```

231. 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 &optional toplevel)
  (do ((form list (cdr form))
       (newform () (cons (walk-form walker (car form) env toplevel)
                         newform)))
      ((atom form) (nreconc newform form))))
```

232. The functions *walk-atomic-form* and *walk-compound-form* are the real work-horses of the walker. Besides the walker instance, the form, and the current environment, *walk-atomic-form* takes a context instance, and *walk-compound-form* takes an *operator* argument like *walk-as-special-form-p*. They both take a *toplevel* keyword argument, which will be true if the form occurs at top level.

```
⟨ Walker generic functions 224 ⟩ +≡
(defgeneric walk-atomic-form (walker form context env &key toplevel))
(defgeneric walk-compound-form (walker operator form env &key toplevel))
```

233. The default method for *walk-atomic-form* simply returns the form.

```
(defmethod walk-atomic-form ((walker walker) form context env &key toplevel)
  (declare (ignore context env toplevel))
  form)
(defmethod walk-atomic-form ((walker walker) (form cons) context env &key toplevel)
  (declare (ignore env toplevel))
  (error "Unexpected␣non-atomic␣form␣~S␣(~S)" form context))
```

234. This method for *walk-compound-form* is used for most funcall-like forms; it treats its car as a function name and walks its cdr.

```
(defmethod walk-compound-form ((walker walker) (operator symbol) form env &key toplevel)
  (declare (ignore toplevel))
  `(,(walk-name walker (car form) (make-context 'operator) env)
    ,@(walk-list walker (cdr form) env)))
```

235. A compound form might also have a λ expression as its car.

```
(deftype lambda-expression () '(cons (eql λ) (cons list *)))
(defmethod walk-compound-form ((walker walker) (operator cons) form env &key toplevel)
  (declare (ignore toplevel))
  (etypecase operator
    (lambda-expression
     `(,(walk-lambda-expression walker operator nil env)
       ,@(walk-list walker (cdr form) env)))))
```

236. Whenever we can recognize a form as a name—a variable name, a function name, *&c.*—we'll walk it using this function. The *context* argument describes the namespace in which the name resides.

237. ⟨ Walker generic functions 224 ⟩ +≡

```
(defgeneric walk-name (walker name context env &key))
```

238. The default method just returns the name being walked.

```
(defmethod walk-name ((walker walker) name context env &key)
  (declare (ignore context env))
  name)
```

239. Walking a name for the purposes of *binding* it in some namespace or the lexical environment is often different, so we'll use another function. The function *walk-bindings* binds each name in *names* in the given *namespace* under the optional control of the declarations in *declare*. It returns two values: the walked names, and a possibly-augmented lexical environment object.

⟨ Walker generic functions 224 ⟩ +≡

```
(defgeneric walk-bindings (walker names namespace env &key declare))
```

240. The default method just walks each of the names and returns the original environment.

```
(defmethod walk-bindings ((walker walker) names (namespace namespace) env &key declare)
  (declare (ignore declare))
  (values (mapcar (λ (name) (walk-name walker name namespace env)) names)
          env))
```

241. As a convenience, we'll use this little function when we only want to walk a single binding.

```
(defun walk-binding (walker name namespace env &key declare)
  (multiple-value-bind (names env)
      (walk-bindings walker (list name) namespace env :declare declare)
    (values (first names) env)))
```

242. 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 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 progv)
  (walk-as-special-form setq)
  (walk-as-special-form unwind-protect))
```

243. The rest of the special form walkers we define will need methods specialized on particular operators 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)
  (let ((oparg `(,(make-symbol "OPERATOR") (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)))))
```

244. *progn* is special because it preserves top-levelness.

```
(define-special-form-walker progn ((walker walker) form env &key toplevel)
  `(,(car form)
    ,@(walk-list walker (cdr form) env toplevel)))
```

245. Block names have their own namespace class.

```
(define-special-form-walker block ((walker walker) form env &key toplevel)
  (declare (ignore toplevel))
  `(,(car form)
    ,(walk-binding walker (cadr form) (make-context 'block-name) env)
    ,@(walk-list walker (cddr form) env)))
(define-special-form-walker return-from ((walker walker) form env &key toplevel)
  (declare (ignore toplevel))
  `(,(car form)
    ,(walk-name walker (cadr form) (make-context 'block-name) env)
    ,(walk-form walker (caddr form) env)))
```

246. *go* tags do, too.

```
(define-special-form-walker tagbody ((walker walker) form env &key toplevel)
  (declare (ignore toplevel))
  `(,(car form)
    ,@(loop with tag-context = (make-context 'tag-name)
            for tag/statement in (cdr form)
            collect (typecase tag/statement
                      ((or symbol integer)
                       (walk-name walker tag/statement tag-context env))
                      (t (walk-form walker tag/statement env))))))
(define-special-form-walker go ((walker walker) form env &key toplevel)
  (declare (ignore toplevel))
  `(,(car form)
    ,(walk-name walker (cadr form) (make-context 'tag-name) env)))
```

247. Ditto for catch tags. Catch tags are evaluated, though, and it's the result that is used as the name, so we'll use a *walk-name* method that actually walks the form. Since catch tags are dynamic, and thus not stored in the lexical environment, we won't use *walk-bindings*, but we will pass a keyword argument to *walk-name* so that specialized methods can discriminate between the establishment of a catch tag and a throw to one.

```
(defmethod walk-name ((walker walker) name (context catch-tag) env &key catch)
  (declare (ignore catch))
  (walk-form walker name env))
(define-special-form-walker catch ((walker walker) form env &key toplevel)
  (declare (ignore toplevel))
  `(,(car form)
    ,(walk-name walker (cadr form) (make-context 'catch-tag) env :catch t)
    ,@(walk-list walker (cddr form) env)))
(define-special-form-walker throw ((walker walker) form env &key toplevel)
  (declare (ignore toplevel))
  `(,(car form)
    ,(walk-name walker (cadr form) (make-context 'catch-tag) env)
    ,(walk-form walker (caddr form) env)))
```

248. The special form *the* takes a type specifier and a form. We won't even bother walking the type specifier. SBCL has a similar but non-standard *truly-the* special form that it unfortunately doesn't provide a macro definition for; we'll support that, too.

```
(defun walk-the (walker form env)
  `(,(car form)
    ,(cadr form)
    ,(walk-form walker (caddr form) env)))
(macrolet ((define-the-walker (operator)
             `(define-special-form-walker ,operator ((walker walker) form env &key toplevel)
                (declare (ignore toplevel))
                (walk-the walker form env))))
  (define-the-walker the)
  #+sbcl (define-the-walker sb-ext:truly-the))
```

249. The *quote* special operator just returns its argument.

```
(define-special-form-walker quote ((walker walker) form env &key toplevel)
  (declare (ignore env toplevel))
  form)
```

250. We'll pretend to be a compiler for the purposes of walking, and evaluate top level forms that appear in an *eval-when* with situation *:compile-toplevel*.

```
(define-special-form-walker eval-when ((walker walker) form env &key toplevel &aux
                                       (eval (and toplevel
                                                  (or (member :compile-toplevel (cadr form))
                                                      (member 'compile (cadr form))))))
  `(,(car form)
    ,(cadr form)
    ,@(loop for form in (cddr form)
            as walked-form = (walk-form walker form env toplevel)
            when eval do (eval walked-form)
            collect walked-form)))
```

251. Allegro's definition of *defconstant* is somewhat funky. Aside from the expected 'special' declamation, the expansion includes calls to two functions: *defconstant1* and *defconstant2* (both in the `EXCL` package). Only the first is wrapped in an (*eval-when* (*compile*)), but it's the second that actually sets the value. The net effect is that if we walk a *defconstant* form in the usual way, the constant gets declaimed special, but remains unbound, which can cause problems.

The work-around is simple: we treat *defconstant* as a special form, and wrap the expanded form in an (*eval-when* (*:compile-toplevel*)). This can't hurt, because the standard says that "[a]n implementation may choose to evaluate the value-form [of a top level *defconstant* form] at compile time, load time, or both."

```
(define-special-form-walker defconstant ((walker walker) form env &key toplevel)
  (declare (ignore toplevel))
  (throw 'continue-walk
    `(eval-when (:compile-toplevel)
       ,(macroexpand-for-walk walker form env))))
```

252. The *function* special form takes either a function name or a λexpression. Under SBCL, this is extended to also include their non-standard *named-lambda* special form, which we'll come to shortly.

```
(deftype named-lambda-expression () '(cons (eql named-lambda)))
(define-special-form-walker function ((walker walker) form env &key toplevel)
  (declare (ignore toplevel))
  `(,(car form)
    ,(typecase (cadr form)
       (lambda-expression (walk-lambda-expression walker (cadr form) nil env))
       (named-lambda-expression (walk-form walker (cadr form) env))
       (t (walk-name walker (cadr form) (make-context 'function-name) env)))))
```

253. 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 false (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 rest)
           (and (stringp x) doc-string-allowed rest (null doc)))
         (declarationp (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 (declarationp x) append (cdr x) into decls
               else do (loop-finish)
          finally (return (values forms
                                  (if walker
                                      (walk-declaration-specifiers walker decls env)
                                      decls)
                                  doc)))))
```

254. Because of the shorthand notation for type declarations, walking general declaration expressions is difficult. However, we don't care about type declarations, since they're not allowed to affect program semantics. We therefore just throw out everything except 'special', 'optimize', 'ignore', and 'ignorable' declarations. Optimize declarations are preserved only because CCL's macro-expansion machinery uses blatantly unsafe code, and depends on local declarations to lower the safety level. Ignore(able) declarations are preserved primarily to keep SBCL from complaining about unused variables in macro functions.

```
(defun walk-declaration-specifiers (walker decls env)
  (loop for decl in decls
        when (walk-declaration-specifier walker decl env) collect it))

(defmethod walk-declaration-specifier ((walker walker) decl-spec env)
  (destructuring-bind (identifier . data) decl-spec
    (case identifier
      (special ⟨ Walk the 'special' declarations in data 255 ⟩)
      ((ignore ignorable) ⟨ Walk the 'ignore' or 'ignorable' declarations in data 256 ⟩)
      (optimize `(optimize ,@data)))))
```

255. ⟨ Walk the 'special' declarations in *data* 255 ⟩ ≡

```
(flet ((walk-var (var)
         (walk-name walker var (make-context 'special-variable-name) env)))
  `(special ,@(mapcar #'walk-var data)))
```

This code is used in section 254.

256. ⟨ Walk the 'ignore' or 'ignorable' declarations in *data* 256 ⟩ ≡

```
(flet ((walk-var/fn (name)
         (etypecase name
           (symbol (walk-name walker name (make-context 'variable-name) env))
           ((cons (eql function)) `(function
                                    ,(walk-name walker (cadr name) (make-context 'function-name) env))))))
  `(,identifier ,@(mapcar #'walk-var/fn data)))
```

This code is used in section 254.

257. ⟨ Walker generic functions 224 ⟩ +≡

```
(defgeneric walk-declaration-specifier (walker decl-spec env))
```

258. Before we dive into λ-lists, let's define our variable-binding walker. We'll return the list of names and an environment augmented with the new bindings.

```
(defmethod walk-bindings ((walker walker) names (namespace variable-name) env &key declare)
  (let ((decls (nconc ⟨ Find declarations in declare that apply to variables in names 259 ⟩
                      (list ⟨ Construct a declaration specifier for special variables in names 260 ⟩))))
    (values names (augment-environment env :variable names :declare decls))))
```

259. SBCL's environment handling is extremely picky, and signals warnings about declarations for variables it doesn't know about. To shut it up, we'll supply only declarations that apply to the variable names currently being added to the environment.

⟨ Find declarations in *declare* that apply to variables in *names* 259 ⟩ ≡

```
(mapcan (λ (decl)
          (and (member (car decl) '(ignore ignorable special))
               (let ((vars (intersection (cdr decl) names)))
                 (and vars (list `(,(car decl) ,@vars))))))
        declare)
```

This code is used in section 258.

260. Lexical bindings in Common Lisp shadow special bindings, *unless* the special declaration is global. However, on all of the implmentations tested so far, *augment-environment always* interprets bindings as lexical unless a *special* declaration is provided, regardless of whether the binding has been proclaimed special in the global environment. This seems to just be a pervasive bug: the specification for *augment-environment* says that "[w]hether each [variable] binding is to be interpreted as special or lexical depends on *special* declarations recorded in the environment or provided in the *:declare* argument" (CLtL-2, p. 212).

We can work around this by checking if a variable was proclaimed special in the global environment, and adding a *special* declaration for the new binding if so. We need to special-case symbols in the `COMMON-LISP` package, because it is an error for a conforming program to declare bindings of such variables *special*. Moreover, SBCL has the notion of a 'package lock' that applies not only to the `COMMON-LISP` package, but to several other implementation-specific packages as well. Symbols in locked packages include the same prohibition on *special* declarations, so we'll omit them from our new declaration as well.

⟨ Construct a declaration specifier for special variables in *names* 260 ⟩ ≡

```
`(special ,@(remove-if-not (λ (name)
                             (and (eq (variable-information name nil) :special)
                                  #+sbcl
                                  (not (sb-ext:package-locked-p (symbol-package name)))
                                  (not (eq (symbol-package name) (find-package "COMMON-LISP")))))
                           names))
```

This code is used in section 258.

261. 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)
  (let ((new-lambda-list '())
        (state :reqvars))
    (labels ((walk-var (var)
               (multiple-value-setq (var env)
                 (walk-binding walker var (make-context 'variable-name :local t) env :declare decls)))
             (update-state (keyword)
               (setq state (case keyword
                             (&optional :optvars)
                             ((&rest &body) :restvars)
                             (&key :keyvars)
                             (&aux :auxvars)
                             (&environment :envvar)
                             (t state))))
             (maybe-destructure (var/pattern)
               (if (consp var/pattern)
                   (multiple-value-setq (var/pattern env)
                     (walk-lambda-list walker var/pattern decls env))
                   (walk-var var/pattern))))
      ⟨ Check for &whole and &environment vars and augment the environment if found 262 ⟩
      (do* ((lambda-list lambda-list (cdr lambda-list))
            (arg (car lambda-list) (if (consp lambda-list) (car lambda-list) lambda-list)))
           ((atom lambda-list)
            (values (nreconc new-lambda-list (and arg (walk-var arg))) env))
        (ecase state
          (:envvar ⟨ Process arg as an environment parameter 263 ⟩)
          ((:reqvars :restvars) ⟨ Process arg as a required parameter 264 ⟩)
          (:optvars ⟨ Process arg as an optional parameter 265 ⟩)
          (:keyvars ⟨ Process arg as a keyword parameter 266 ⟩)
          (:auxvars ⟨ Process arg as an auxiliary variable 267 ⟩))))))
```

262. 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 environment if found 262 ⟩ ≡

```
(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)))
(do ((lambda-list lambda-list (cdr lambda-list)))
    ((atom lambda-list) nil)
  (when (eql (car lambda-list) '&environment)
    (return (cadr lambda-list))))
```

This code is used in section 261.

263. 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 263 ⟩ ≡
(push (walk-var arg) new-lambda-list)
(when (consp lambda-list)
  (update-state (car lambda-list)))
```

This code is used in section 261.

264. Required parameters may be either symbols or patterns. Rest parameters have the same syntax.

```
⟨ Process arg as a required parameter 264 ⟩ ≡
(etypecase arg
  (symbol ⟨ Process the symbol in arg as a parameter 268 ⟩)
  (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 261.

265. 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 265 ⟩ ≡
(etypecase arg
  (symbol ⟨ Process the symbol in arg as a parameter 268 ⟩)
  (cons
   (destructuring-bind (var/pattern &optional
                        (init-form nil init-form-supplied)
                        (supplied-p-parameter nil spp-supplied))
       arg
     (when init-form-supplied
       (setq init-form (walk-form walker init-form env)))
     (push (nconc (list (maybe-destructure var/pattern))
                  (and init-form-supplied (list init-form))
                  (and spp-supplied (list (walk-var supplied-p-parameter))))
           new-lambda-list))))
```

This code is used in section 261.

266. ⟨ Process *arg* as a keyword parameter 266 ⟩ ≡

```
(etypecase arg
  (symbol ⟨ Process the symbol in arg as a parameter 268 ⟩)
  (cons
   (destructuring-bind (var/kv &optional
                        (init-form nil init-form-supplied)
                        (supplied-p-parameter nil spp-supplied))
       arg
     (when init-form-supplied
       (setq init-form (walk-form walker init-form env)))
     (cond ((consp var/kv)
            (destructuring-bind (keyword-name var/pattern) var/kv
              (setq var/pattern (maybe-destructure var/pattern))
              (setq var/kv (list (walk-atomic-form walker keyword-name nil env)
                                 var/pattern))))
           (t (setq var/kv (walk-var var/kv))))
     (push (nconc (list var/kv)
                  (and init-form-supplied (list init-form))
                  (and spp-supplied (list (walk-var supplied-p-parameter))))
           new-lambda-list))))
```

This code is used in section 261.

267. ⟨ Process *arg* as an auxiliary variable 267 ⟩ ≡

```
(etypecase arg
  (symbol ⟨ Process the symbol in arg as a parameter 268 ⟩)
  (cons
   (destructuring-bind (var &optional init-form) arg
     (setq var (walk-var var)
           init-form (and init-form (walk-form walker init-form env)))
     (push (nconc (list var)
                  (and init-form (list init-form)))
           new-lambda-list))))
```

This code is used in section 261.

268. ⟨ Process the symbol in *arg* as a parameter 268 ⟩ ≡

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

This code is used in sections 264, 265, 266, and 267.

269. 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 270 ⟩))
    (multiple-value-bind (other-params env) (walk-lambda-list walker lambda-list decls env)
      (values (nconc req-params other-params) env))))
```

270. ⟨ Extract the required parameters from *lambda-list* 270 ⟩ ≡

```
(flet ((walk-var (spec &aux (context (make-context 'variable-name :local t)))
         (flet ((walk-binding (x &aux name)
                  (declare (ignorable name)) ; Allegro thinks this is ignored
                  (multiple-value-setq (name env)
                    (walk-binding walker x context env :declare decls))))
           (etypecase spec
             (symbol (walk-binding spec))
             (class-specializer
              (list (walk-binding (car spec))
                    (walk-name walker (cadr spec) (make-context 'class-name%) env)))
             ((compound-specializer eql)
              (list (walk-binding (car spec))
                    `(eql ,(walk-form walker (cadadr spec)))))))))
  (loop until (or (null lambda-list)
                  (member (car lambda-list) lambda-list-keywords))
        collect (walk-var (pop lambda-list))))
```

This code is used in section 269.

271. Having built up the necessary machinery, walking a λ expression is now straightforward. The slight generality of possibly walking the car of the form using *walk-name* is because this function will also be used to walk the bindings in *flet*, *macrolet*, and *labels* special forms.

```
(defun walk-lambda-expression (walker form context env)
  (let ((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)
        `(,(let ((name (car form)))
             (case name
               (λ name)
               (t (walk-name walker (car form) context env))))
          ,lambda-list
          ,@(when doc `(,doc))
          ,@(when decls `((declare ,@decls)))
          ,@(walk-list walker forms env))))))
```

272. 'λ' 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 &key toplevel)
  (declare (ignore toplevel))
  (walk-lambda-expression walker form nil env))
```

273. We also support a 'named lambda' form. Several Lisp implementations use such forms internally, but SBCL does not provide a macro defintion for theirs. This is just rude; Allegro has a similar operator called *named-function*, but they provide a macro definition that expands into an ordinary λ expression. The syntax is assumed to be '(*named-lambda* ⟨*name*⟩ ⟨*lambda-list*⟩ ⟨*body*⟩)'.

```
(define-special-form-walker named-lambda ((walker walker) form env &key toplevel)
  (declare (ignore toplevel))
  (walk-lambda-expression walker `(λ ,(caddr form) ,@(cdddr form)) nil env))
```

274. We come now to the binding special forms. The six lexical binding 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.

275. *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 &key toplevel)
  (declare (ignore toplevel))
  (multiple-value-bind (forms decls) (parse-body (cddr form) :walker walker :env env)
    (let* ((bindings (mapcar #'ensure-list (cadr form)))
           (init-forms (mapcar (λ (form) (walk-form walker form env))
                               (mapcar #'cadr bindings))))
      (multiple-value-bind (vars env)
          (walk-bindings walker
                         (mapcar #'car bindings)
                         (make-context 'variable-name :local t)
                         env
                         :declare decls)
        `(,(car form)
          ,(mapcar #'list vars init-forms)
          ,@(when decls `((declare ,@decls)))
          ,@(walk-list walker forms env))))))
```

276. We need a *walk-bindings* method for local function bindings; this will be used for *labels* as well. Notice that we don't currently preserve any declarations that can affect function bindings, so we simply ignore any supplied declarations.

```
(defmethod walk-bindings ((walker walker) names (namespace function-name) env &key declare)
  (declare (ignore declare))
  (values names
          (augment-environment env :function names)))
```

277.

```
(define-special-form-walker flet ((walker walker) form env &key toplevel)
  (declare (ignore toplevel))
  (multiple-value-bind (forms decls) (parse-body (cddr form) :walker walker :env env)
    (let* ((bindings (cadr form))
           (context (make-context 'function-name :local t))
           (fns (mapcar (λ (fn) (walk-lambda-expression walker fn context env))
                        bindings)))
      (multiple-value-bind (function-names env)
          (walk-bindings walker
                         (mapcar #'car bindings)
                         context
                         env
                         :declare decls)
        `(,(car form)
          ,(mapcar #'cons function-names (mapcar #'cdr fns))
          ,@(when decls `((declare ,@decls)))
          ,@(walk-list walker forms (if decls
                                        (augment-environment env :declare decls)
                                        env)))))))
```

278. 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 need to build the expander functions using *parse-macro* and *enclose* before we add them to the environment.

```
(defmethod walk-bindings ((walker walker) defs (namespace macro-definition) env &key declare)
  (let* ((fns (mapcar (λ (exp) (walk-lambda-expression walker exp nil env))
                      defs))
         (defs (mapcar (λ (def)
                         (destructuring-bind (name lambda-list &rest body) def
                           (list name
                                 (enclose (parse-macro name lambda-list body env) env walker))))
                       fns)))
    (values fns (augment-environment env :macro defs :declare declare))))
```

279. Note that *macrolet* preserves top-levelness of its body forms.

```
(define-special-form-walker macrolet ((walker walker) form env &key toplevel)
  (multiple-value-bind (forms decls) (parse-body (cddr form) :walker walker :env env)
    (let ((bindings (cadr form)))
      (multiple-value-bind (defs env)
          (walk-bindings walker bindings
                         (make-context 'macro-definition :local t)
                         env
                         :declare decls)
        `(,(car form)
          ,defs
          ,@(when decls `((declare ,@decls)))
          ,@(walk-list walker forms env toplevel))))))
```

280. Walking *symbol-macrolet* is simpler, since the definitions are given in the bindings themselves. The body forms of a top level *macrolet* form are also considered top level.

```
(defmethod walk-bindings ((walker walker) defs (namespace symbol-macro-definition) env &key declare)
  (values (setq defs (mapcar (λ (p)
                               `(,(car p)
                                 ,(walk-form walker (cadr p) env)))
                             defs))
          (augment-environment env :symbol-macro defs :declare declare)))
(define-special-form-walker symbol-macrolet ((walker walker) form env &key toplevel)
  (multiple-value-bind (forms decls) (parse-body (cddr form) :walker walker :env env)
    (let ((bindings (cadr form)))
      (multiple-value-bind (defs env)
          (walk-bindings walker bindings
                         (make-context 'symbol-macro-definition :local t)
                         env
                         :declare decls)
        `(,(car form)
          ,defs
          ,@(when decls `((declare ,@decls)))
          ,@(walk-list walker forms env toplevel))))))
```

281. 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 &key toplevel)
  (declare (ignore toplevel))
  (multiple-value-bind (forms decls) (parse-body (cddr form) :walker walker :env env)
    (let ((context (make-context 'variable-name :local t)))
      `(,(car form)
        ,(mapcar (λ (p &aux (p (ensure-list p)))
                   (let ((var (car p))
                         (init-form (walk-form walker (cadr p) env)))
                     (multiple-value-setq (var env)
                       (walk-binding walker var context env :declare decls))
                     (list var init-form)))
                 (cadr form))
        ,@(when decls `((declare ,@decls)))
        ,@(walk-list walker forms env)))))
```

282.

```
(define-special-form-walker labels ((walker walker) form env &key toplevel)
  (declare (ignore toplevel))
  (multiple-value-bind (forms decls) (parse-body (cddr form) :walker walker :env env)
    (let* ((context (make-context 'function-name :local t))
           (bindings (cadr form))
           (function-names '()))
      (dolist (binding bindings (setf function-names (nreverse function-names)))
        (multiple-value-bind (function-name new-env)
            (walk-binding walker (car binding) context env :declare decls)
          (push function-name function-names)
          (setq env new-env)))
      `(,(car form)
        ,(mapcar (λ (p) (walk-lambda-expression walker p context env))
                 (mapcar #'cons function-names (mapcar #'cdr bindings)))
        ,@(when decls `((declare ,@decls)))
        ,@(walk-list walker forms (if decls
                                      (augment-environment env :declare decls)
                                      env))))))
```

283. 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. Note that *locally* preserves top-levelness of its body forms.

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

284. In order to recognize global proclamations, we'll treat *declaim* as a special form.

```
(define-special-form-walker declaim ((walker walker) form env &key toplevel)
  `(,(car form)
    ,@(let ((decls (walk-declaration-specifiers walker (cdr form) env)))
        (when toplevel
          (mapcar #'proclaim decls))
        decls)))
```

285. Indexing. Formally, an *index* is an ordered collection of *entries*, each of which is a (⟨*heading*⟩, ⟨*locator*⟩) pair: the *locator* indicates where the object denoted 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.

In this program, our main job in indexing is to automatically produce entries whose headings are names (symbols, generally) together with the namespace in which the name is bound, and whose locators are the sections in which that binding is defined or used. E.g., given a definition of a function named *foo* in some section m and a use of that function in section n, we'd like to produce an entry list like '*foo* function: $\underline{m}, n$'. If there were also a class named *foo*, that would have a separate entry.

The basic idea is to use a specialized walker class to perform a code walk of all the code parts in a web, and to build the index as we go. Because the code walker knows about all of the standard namespaces and how the various special forms introduce new bindings, we can use the context it provides during a walk to construct our headings. We'll have to use a trick to get the locations right, but we'll come to that in a bit.

286. Of course, we don't want to index *all* of the symbols in a web: nobody really cares about all of the uses of *let*, for instance—or indeed any (or at least most) of the symbols in the `COMMON-LISP` package. Instead, we'll have the notion of an *interesting* symbol; if a symbol is interesting, we'll index it, and if not, then we won't.

By default, we'll say a symbol is interesting if its home package in one of the packages listed in ∗*index-packages*∗. By adding methods to the GF *interesting-symbol-p*, the user can extend this notion in essentially arbitrary ways.

```
(defgeneric interesting-symbol-p (object))
(defmethod interesting-symbol-p (object)
  (declare (ignore object)))
(defmethod interesting-symbol-p ((object symbol))
  (member (symbol-package object) ∗index-packages∗))
```

287. ⟨ Global variables 12 ⟩ +≡

```
(defvar ∗index-packages∗ nil
  "The␣list␣of␣packages␣whose␣symbols␣should␣be␣indexed.")
```

288. ⟨ Initialize global variables 25 ⟩ +≡

```
(setq ∗index-packages∗ nil)
```

289. The user can add packages to the ∗*index-packages*∗ list using the `@x` control code, which calls the following function.

```
(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∗)))
```

290. Headings. In this program, headings will generally be represented by instances of the class *heading*. Headings may in general be multi-leveled, and are sorted lexicographically. Any object with applicable methods for *heading-name* and *sub-heading* is treated as a valid heading; the former should always return a string designator, and the latter should return the next sub-heading or *nil* if there is none.

⟨ Define *heading-name* generic function 294 ⟩

```
(defclass heading ()
  ((name :initarg :name :initform "")
   (sub-heading :reader sub-heading :initarg :sub-heading :initform nil)))
(defun make-heading (name &optional sub-heading)
  (make-instance 'heading :name name :sub-heading sub-heading))
(defmethod heading-name ((heading heading))
  (heading-name (slot-value heading 'name)))
(defmethod heading-name :suffix ((heading heading))
  (when (sub-heading heading)
    (heading-name (sub-heading heading))))
```

291. These heading classes are for headings that should be prefaced with a TEX macro when printed. Instances of *tt-heading* get printed with '`\.`' so as to appear in `typewriter type`, and *custom-heading* instances are printed under the control of the TEX macro '`\9`', which the user can define as desired. Additional subclasses can simply provide new defaults for the *macro* initarg.

```
(defclass pretty-heading (heading)
  ((macro :reader macro-heading :initarg :macro)))
(defclass tt-heading (pretty-heading) () (:default-initargs :macro "\\."))
(defclass custom-heading (pretty-heading) () (:default-initargs :macro "\\9"))
```

292. We'll allow string designators as headings, too. Notice that we strip leading whitespace and backslashes from strings acting as heading names; this helps the common case of a manual entry that begins with a TEX macro sort correctly.

```
(defmethod sub-heading (heading)
  (declare (ignore heading)))
(defmethod heading-name ((heading character))
  heading)
(defmethod heading-name ((heading string))
  (string-left-trim '(#\␣ #\Tab #\Newline #\\) heading))
(defmethod heading-name ((heading symbol))
  heading)
```

293. We'll be storing index entries in a BST ordered by heading, so we'll need some comparison predicates for them. These are generic functions so that the user may provide specialized methods on their own heading classes if they want them sorted in a particular way. Letter case is ignored by default.

```
(defgeneric entry-heading-lessp (h1 h2))
(defmethod entry-heading-lessp (h1 h2)
  (string-lessp (heading-name h1) (heading-name h2)))
(defgeneric entry-heading-equalp (h1 h2))
(defmethod entry-heading-equalp (h1 h2)
  (string-equal (heading-name h1) (heading-name h2)))
```

294. When we build entries for Lisp objects, we'll use the walker's namespace objects as sub-headings. In order to make creating nice names from those objects easier, we'll use a custom method combination that joins together the results of each applicable method into one delimited string. This turns out to be generally convenient for heading names of all kinds, so we'll use it for *heading-name*.

⟨ Define *heading-name* generic function 294 ⟩ ≡

```
(defun join-strings (strings &optional (delimiter #\␣) &aux
                     (strings (ensure-list strings))
                     (delimiter (string delimiter)))
  (with-output-to-string (out)
    (loop for (string . more) on strings
          when string
           do (write-string (string string) out)
           and when more do (write-string delimiter out))))
(define-method-combination join-strings (&optional (delimiter #\␣))
    ((override (:override))
     (prefix (:prefix))
     (primary () :required t)
     (suffix (:suffix)))
  (flet ((call-methods (methods)
           (mapcar (λ (method) `(ensure-list (call-method ,method))) methods)))
    (let ((form `(join-strings (append ,@(call-methods prefix)
                                       ,@(call-methods primary)
                                       ,@(call-methods (reverse suffix)))
                               ,delimiter)))
      (if override
          `(call-method ,(first override) (,@(rest override) (make-method ,form)))
          form))))
(defgeneric heading-name (heading)
  (:method-combination join-strings))
```

This code is used in section 290.

295. Every namespace has a name associated with it, and we'll derive our primary heading name from that.

```
(defmethod heading-name ((namespace namespace))
  (substitute #\␣ #\- (string-downcase (namespace-name namespace))))
```

296. Lexical and special variables need their names to be *:lexical* and *:special*, respectively, to match the types returned by the environment functions. We'll append the suffix "variable" to both.

```
(defmethod heading-name :suffix ((namespace lexical-variable-name)) "variable")
(defmethod heading-name :suffix ((namespace special-variable-name)) "variable")
```

297. Local functions, macros, and symbol macros should be marked as such.

```
(defmethod heading-name :prefix ((namespace operator))
  (and (local-binding-p namespace) "local"))
(defmethod heading-name :prefix ((namespace symbol-macro-name))
  (and (local-binding-p namespace) "local"))
```

298. Method definitions will have their qualifiers prepended, or 'primary' if there are none.

```
(defmethod heading-name :prefix ((namespace method-name))
  (mapcar #'string-downcase (or (method-qualifier-names namespace) '(:primary))))
```

299. Locators. 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 :initform nil)))
(defclass xref-locator (locator)
  ((heading :accessor location :initarg :heading)))
(defclass see-locator (xref-locator) ())
(defclass see-also-locator (xref-locator) ())
```

300. 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))))
```

301. Index entries. 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))
```

302. We'll store the entry trees in *index* objects.

```
(defclass index ()
  ((entries :accessor index-entries :initform nil)))
(defun make-index () (make-instance 'index))
```

303. 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)
```

304.

```
⟨ Initialize global variables 25 ⟩ +≡
(setq *index* (make-index))
```

305. This function 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)
(defun add-index-entry (index heading section &optional def)
  (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)))))))
```

306. And this function looks up a specific heading in an index, and returns any locators that it finds.

```
(defun find-index-entries (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))))))
```

307. Now we come to the interface between the indexer and the indexing walker we'll define below. The idea is that the walker picks up information about symbols and function names and in what section they're defined or used, and passes it all down to *index* encoded in the *context* object.

```
(defgeneric index (index name section context &optional def))
```

```
(defmethod index ((index index) name section context &optional def)
  (when (and name section context)
    (add-index-entry index (make-heading name context) section def)))
```

308. The global variable ∗*index-lexical-variables*∗ controls whether or not the indexer will create entries for lexical variables. Its value is *not* re-initialized on each run.

```
⟨ Global variables 12 ⟩ +≡
(defvar ∗index-lexical-variables∗ nil
  "If␣this␣flag␣is␣non-nil,␣the␣indexer␣will␣index␣lexical␣variables.")
```

309.

```
(defmethod index ((index index) name section (context lexical-variable-name) &optional def)
  (declare (ignore name section def))
  (when ∗index-lexical-variables∗
    (call-next-method)))
```

310. We index special and (sometimes) lexical variables, but we don't want to index undifferentiated variable names (e.g., as occur in declarations, *&c.*).

```
(defmethod index :around ((index index) name section (context variable-name) &optional def)
  (declare (ignore name section def))
  (unless (eq (class-of context) (find-class 'variable-name))
    (call-next-method)))
```

311. There are a few other kinds of names that we walk but don't currently bother indexing.

```
(defmacro dont-index (namespace)
  `(defmethod index :around ((index index) name section (context ,namespace) &optional def)
     (declare (ignore name section def))))
(dont-index block-name)
(dont-index tag-name)
(dont-index slot-name)
```

312. Referring symbols. 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 rôle. 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 call 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. We take care to substitute inside of commas, too.

```
(defun substitute-referring-symbols (form section)
  (let (symbols refsyms)
    (labels ((collect-symbols (form)
               (cond ((interesting-symbol-p form)
                      (pushnew form symbols))
                     ((atom form) nil)
                     (t (collect-symbols (car form))
                        (collect-symbols (cdr form)))))
             (make-referring-symbol (symbol)
               (let ((refsym (copy-symbol symbol)))
                 (setf (symbol-value refsym) symbol)
                 (setf (get refsym 'section) section)
                 ⟨ Copy any macro function associated with symbol to refsym 313 ⟩
                 ⟨ Make refsym an alias for any class named by symbol 314 ⟩
                 (cons symbol refsym)))
             (substitute-symbols (form)
               (cond ((commap form)
                      (make-comma (comma-modifier form)
                                  (substitute-referring-symbols (comma-form form) section)))
                     ((symbolp form)
                      (or (cdr (assoc form refsyms)) form))
                     ((atom form) form)
                     (t (maptree #'substitute-symbols form)))))
      (collect-symbols form)
      (setq refsyms (mapcar #'make-referring-symbol symbols))
      (maptree #'substitute-symbols form))))
```

313. If the original symbol has a global definition as a macro, we'll copy that, too, so as not to break code walkers and such.

```
⟨ Copy any macro function associated with symbol to refsym 313 ⟩ ≡
(let ((function (macro-function symbol)))
  (when function
    (setf (macro-function refsym) function)))
```

This code is used in section 312.

314. And if the original symbol names a class, we'll make the referring symbol an alias for that class. This is necessary, for instance, when a referring symbol replaces a symbol naming a class in a type specification that is used before we have a chance to swap it back out.

The association between symbols and classes named by those symbols is not stored in the symbols themselves, which means that we need to manually remove the associations when we no longer need them. So whenever we alias a class with a referring symbol, we'll push that symbol onto the global ∗*referring-classes*∗ list, and we'll remove the association when we're finished indexing.

⟨ Make *refsym* an alias for any class named by *symbol* 314 ⟩ ≡
```
(let ((class (find-class symbol nil)))
  (when class
    (setf (find-class refsym) class)
    (push refsym ∗referring-classes∗)))
```
This code is used in section 312.

315. ⟨ Global variables 12 ⟩ +≡
```
(defvar ∗referring-classes∗ '())
```

316. ⟨ Initialize global variables 25 ⟩ +≡
```
(setq ∗referring-classes∗ '())
```

317. Unfortunately, on most implementations simply removing the association between a symbol and a class is not enough to actually free the storage allocated to that association. Since we're potentially generating many such associations (hundreds are created during the indexing of this program, for instance), this could leak non-trivial amounts of memory, especially with repeated runs.

⟨ Clean up after indexing 317 ⟩ ≡
```
(dolist (symbol ∗referring-classes∗ (setq ∗referring-classes∗ '()))
  (setf (find-class symbol) nil))
```
This code is used in section 323.

318. Given a symbol, this next function first determines whether it is a referring symbol, and if so, it returns the referenced symbol and the section from whence it came. 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)
    (if (and (not (symbol-package symbol))
             (boundp symbol)
             (setq section (get symbol 'section)))
        (values (symbol-value symbol) section)
        symbol)))
```

319. To replace all of the referring symbols in a form, we'll use the following simple function.

```
(defun unsubstitute-referring-symbols (x)
  (typecase x
    (symbol (symbol-provenance x))
    (atom x)
    (t (maptree #'unsubstitute-referring-symbols x))))
```

320. 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-referring-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-referring-symbols (tangle code :expand-named-sections nil) section)
        code)))
```

321. 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))))
```

322. ⟨ Global variables 12 ⟩ +≡

```
(defvar ∗indexing∗ nil)
```

323. So 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))
    (index-package ∗package∗)
    (unwind-protect
         (dolist (form (tangle-code-for-indexing sections) (walker-index walker))
           (handler-case (walk-form walker form nil t)
             (package-error () form)))
      ⟨ Clean up after indexing 317 ⟩)))
```

324. The indexing walker. Now we're ready to define the specialized walker class that does the actual indexing.

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

325. We'll start with a method that undoes the substitution of referring symbols inside quoted forms.

```
(defmethod walk-compound-form ((walker indexing-walker) (operator (eql 'quote)) form env &key toplevel)
  (declare (ignore env toplevel))
  `(quote ,(unsubstitute-referring-symbols (cadr form))))
```

326. We have to override the walker's macro expansion function, since the forms that we're considering might be or contain referring symbols, which might not have the appropriate definition as macros. Moreover, we ignore all errors that may occur during expansion, since not all macros are well-behaved in the presence of referring symbols.

```
(defmethod macroexpand-for-walk ((walker indexing-walker) (form cons) env)
  (handler-case ⟨ Index an expanded form 327 ⟩
    (error () form)))
```

327. There are two important cases here: (1) a form that is a referring symbol whose referent is a symbol macro in the local or global environment; and (2) a compound form, the operator of which is a referring symbol whose referent has a macro definition in the local environment. The first case is handled via the call to *walk-variable-name* in *walk-form* for symbol macros. We handle the second case here: we'll index the use of the macro, then hand off control to the next method (which will perform the actual expansion), passing the referent of the referring symbol.

```
⟨ Index an expanded form 327 ⟩ ≡
(multiple-value-bind (symbol section) (symbol-provenance (car form))
  (if section
      (multiple-value-bind (type local) (function-information symbol env)
        (case type
          (:macro
           (index (walker-index walker) symbol section (make-context 'macro-name :local local))
           (call-next-method walker (cons symbol (cdr form)) env))
          (t (call-next-method))))
      (call-next-method)))
```

This code is used in section 326.

328. The only ordinary atomic forms we care about are referring symbols, which we'll index and then return the referents of. Everything else gets handled by the default method.

```
(defmethod walk-atomic-form ((walker indexing-walker) (form symbol) context env &key toplevel)
  (declare (ignore env toplevel))
  (multiple-value-bind (symbol section) (symbol-provenance form)
    (when section
      (index (walker-index walker) symbol section context))
    symbol))
```

329. To provide a hook for indexing arbitrary function calls, we'll swap out referring symbols that occur as the operator in a compound form. This lets us write *eql*-specialized methods on the *operator* argument while still preserving the provenance via the *form*.

If the operator is a referring symbol, we have to do a full call to pick up any newly-applicable methods. Otherwise, we can just use *call-next-method*.

```
(defmethod walk-compound-form :around ((walker indexing-walker) (operator symbol) form env &rest args)
  (multiple-value-bind (symbol section) (symbol-provenance operator)
    (if section
      (apply #'walk-compound-form walker symbol form env args)
      (call-next-method))))
```

330. When we walk a name that is or contains a referring symbol, we'll swap out the symbol and index the name. This method handles the mechanics by using two auxiliary routines, *destructure-name* and *construct-name*. The idea is that a name might be as simple as a symbol, or a list like '(*setf* ⟨*name*⟩)', or, as in the case of a macro definition, even include a definition. *destructure-name* takes apart such names based on the namespace and returns the symbol we'll use in the index if it's a referring symbol; *construct-name* goes the other way.

```
(defgeneric destructure-name (name namespace))
(defgeneric construct-name (symbol name namespace))
```

```
(defmethod walk-name ((walker indexing-walker) name namespace env &key def)
  (multiple-value-bind (symbol section) (symbol-provenance (destructure-name name namespace))
    (let ((name (construct-name symbol name namespace)))
      (when section
        (index (walker-index walker) symbol section
               (if def namespace (update-context name namespace env))
               def))
      name)))
```

331. We'll use the same auxiliary routines for walking bindings. It's slightly complicated because we have to be able to handle multiple bindings simultaneously, and we need to wait for the indexing until we have the augmented environment. So we hold on to the referred-to symbols and their provenances while we call up to the next method to perform the actual bindings, and then index them one at a time.

```
(defmethod walk-bindings ((walker indexing-walker) names namespace env &key declare)
  (let ((symbols '())
        (sections '()))
    (dolist (name names (progn (setf symbols (nreverse symbols))
                               (setf sections (nreverse sections))))
      (multiple-value-bind (symbol section) (symbol-provenance (destructure-name name namespace))
        (push symbol symbols)
        (push section sections)))
    (multiple-value-bind (names env)
        (call-next-method walker
                          (mapcar (λ (symbol name) (construct-name symbol name namespace))
                                  symbols names)
                          namespace env
                          :declare declare)
      (loop for symbol in symbols and section in sections and name in names
            when section
              do (index (walker-index walker) symbol section (update-context name namespace env) t))
      (values names env))))
```

332. Here are the methods for destructuring names. The default method is the common case, where no destructuring is actually necessary. Functions are either symbols or *setf* functions, and macro and symbol macro definitions are represented using lists of the form '(⟨*name*⟩ ⟨*definition*⟩)'.

```
(defmethod destructure-name (name namespace)
  (declare (ignore namespace))
  name)

(defmethod destructure-name (name (namespace function-name))
  (etypecase name
    (symbol name)
    (setf-function (cadr name))))

(defmethod destructure-name (def (namespace macro-definition))
  (car def))

(defmethod destructure-name (def (namespace symbol-macro-definition))
  (car def))
```

333. And here's how we put names back together based on their original form.

```
(defmethod construct-name (symbol name namespace)
  (declare (ignore name namespace))
  symbol)

(defmethod construct-name (symbol name (namespace function-name))
  (etypecase name
    (symbol symbol)
    (setf-function `(setf ,symbol))))

(defmethod construct-name (symbol def (namespace macro-definition))
  (cons symbol (cdr def)))

(defmethod construct-name (symbol def (namespace symbol-macro-definition))
  (cons symbol (cdr def)))
```

334. The one kind of name we'll walk differently is catch tags. Because they might be arbitrary forms, we can't reliably specify how to take them apart and put them back together. But that doesn't matter, because we only care about one specific kind of tag: quoted symbols.

```
(deftype quoted-symbol () '(cons (eql quote) (cons symbol null)))

(defmethod walk-name ((walker indexing-walker) tag (namespace catch-tag) env &key catch)
  (typecase tag
    (quoted-symbol
     (multiple-value-bind (symbol section) (symbol-provenance (cadr tag))
       (when section
         (index (walker-index walker) symbol section namespace catch))
       `(quote ,symbol)))
    (t (walk-form walker tag env))))
```

335. Indexing defining forms. Now we can turn our attention to walking defining forms such as *defun* and *defparameter*. The idea here is to index the name being defined before macro expansion robs us of the opportunity. We'll do that by pretending that the defining forms are actually special forms, which stops the walker from expanding them. After we're finished indexing, we can let the expansion continue via a *throw* to the *continue-walk* tag. But sometimes we won't even bother with the expansions; we don't care about getting an accurate and complete walk here, but rather only about indexing all of the interesting names.

336. *defun* and *define-compiler-macro* are perfect examples of defining forms that we'll walk as special forms for the purposes of indexing. If we didn't catch them before macro expansion, we'd have a very hard time noticing that they were definitions at all. We won't bother letting them expand, since we get everything we need from walking the name with *walk-name* and the body with *walk-lambda-expression*.

```
(defun walk-defun (walker form context env)
  (let ((name (walk-name walker (cadr form) context env :def t)))
    `(,(car form)
      ,@(walk-lambda-expression walker (cons name (cddr form)) context env))))
(define-special-form-walker defun ((walker indexing-walker) form env &key toplevel)
  (declare (ignore toplevel))
  (walk-defun walker form (make-context 'function-name) env))
(define-special-form-walker define-compiler-macro ((walker indexing-walker) form env &key toplevel)
  (declare (ignore toplevel))
  (walk-defun walker form (make-context 'compiler-macro-name) env))
```

337. We'll handle *defmacro* and *define-symbol-macro* similarly, except that we do need them to be expanded so that we can pick up their definitions in the global environment.

```
(define-special-form-walker defmacro ((walker indexing-walker) form env &key toplevel)
  (declare (ignore toplevel))
  (throw 'continue-walk
    (walk-defun walker form (make-context 'macro-name) env)))
(define-special-form-walker define-symbol-macro ((walker indexing-walker) form env &key toplevel)
  (declare (ignore toplevel))
  (let* ((context (make-context 'symbol-macro-name))
         (name (walk-name walker (cadr form) context env :def t)))
    (throw 'continue-walk
      `(,(car form) ,name ,(walk-form walker (caddr form) env)))))
```

338. We'll need to let *defvar* and *defparameter* expand after walking, too, in order to pick up the *special* proclamations.

```
(defun walk-defvar (walker form env)
  (throw 'continue-walk
    `(,(car form)
      ,(walk-name walker (cadr form) (make-context 'special-variable-name) env :def t)
      ,@(cddr form))))
(define-special-form-walker defvar ((walker indexing-walker) form env &key toplevel)
  (declare (ignore toplevel))
  (walk-defvar walker form env))
(define-special-form-walker defparameter ((walker indexing-walker) form env &key toplevel)
  (declare (ignore toplevel))
  (walk-defvar walker form env))
```

339. We have to treat *defconstant* specially because of the work-around in the walker for Allegro, but it's basically the same as above.

```
(define-special-form-walker defconstant ((walker indexing-walker) form env &rest args)
  (apply #'call-next-method walker 'defconstant
         `(defconstant ,(walk-name walker (cadr form) (make-context 'constant-name) env :def t)
            ,@(walk-list walker (cddr form) env))
         env args))
```

340. Structure definitions get walked to pick up the structure name as well as any automatically defined functions. We'll let the walk continue with the expansion so that the compiler can pick up the structure definition, too.

```
⟨ Declare structure-description structure 341 ⟩
⟨ Define defstruct parsing routines 342 ⟩
(define-special-form-walker defstruct ((walker indexing-walker) form env &key toplevel)
  (declare (ignore toplevel))
  (let ((structure-description)
        (name-and-options))
    (declare (ignorable name-and-options)) ; for Allegro and CCL
    (throw 'continue-walk
      `(,(pop form)
        ,(multiple-value-setq (name-and-options structure-description)
                              (walk-defstruct-name-and-options walker (pop form) env))
        ,@(and (stringp (car form)) (list (pop form)))
        ,@(mapcar (λ (slot-description)
                    (walk-defstruct-slot-description walker slot-description structure-description env))
                  form)))))
```

341. We'll use an auxiliary data structure, called a *structure description*, to handle most of the bookkeeping as we walk a *defstruct* form. (Some of this code was inspired by SBCL's *defstruct* processing, although it is simplified in many ways.)

```
⟨ Declare structure-description structure 341 ⟩ ≡
(defun symbolicate (&rest strings)
  (values (intern (join-strings strings ""))))
(defstruct (structure-description
            (:conc-name "STRUCT-")
            (:constructor make-structure-description
                          (name &aux
                           (conc-name (symbolicate name "-"))
                           (copier-name (symbolicate "COPY-" name))
                           (predicate-name (symbolicate name "-P")))))
  (name nil :type symbol :read-only t)
  (conc-name nil :type (or symbol null))
  (constructors () :type list)
  (copier-name nil :type (or symbol null))
  (predicate-name nil :type (or symbol null))
  (include nil :type list))
```

This code is used in section 340.

342. We'll need several subroutines to do the work of parsing and walking a *defstruct* form. We'll start with the processing of the name and options.

Notice that *walk-defstruct-name-and-options* returns two values: the walked name and options (a form), and the structure description that will be passed down to the slot description parsing routines.

The recursion in *walk-defstruct-option* is just a lazy way of canonicalizing options supplied as atoms that are equivalent to one-element lists; it will never go more than one level deep.

⟨ Define *defstruct* parsing routines 342 ⟩ ≡

```
(defun walk-defstruct-name-and-options (walker name-and-options env)
  (destructuring-bind (name &rest options) (ensure-list name-and-options)
    (let* ((struct-name (walk-name walker name (make-context 'struct-name) env :def t))
           (struct-description (make-structure-description struct-name)))
      (values `(,struct-name
                ,@(loop for option in options
                        collect (walk-defstruct-option walker option struct-description env)
                        finally ⟨ Index auto-generated structure function names 344 ⟩))
              struct-description))))

(defun walk-defstruct-option (walker option description env)
  (cond ((member option '(:conc-name :constructor :copier :predicate))
         (walk-defstruct-option walker (list option) description env))
        ((atom option)
         option)
        (t `(,(first option)
             ,@(ensure-list ⟨ Walk one defstruct option 343 ⟩)))))
```

See also sections 345 and 346.

This code is used in section 340.

343. For the purposes of indexing, we only care about a handful of *defstruct* options. At this point, *option* will have been converted to list form; we'll walk it and record relevant information in *description* as we go.

⟨ Walk one *defstruct* option 343 ⟩ ≡

```
(let ((args (rest option))
      (struct-name (struct-name description)))
  (case (first option)
    (:conc-name
     (destructuring-bind (&optional conc-name) args
       (setf (struct-conc-name description)
             (if (symbolp conc-name)
                 (walk-atomic-form walker conc-name nil env)
                 (make-symbol (string conc-name))))))
    (:constructor
     (destructuring-bind (&optional (name (symbolicate "MAKE-" struct-name)) &rest arglist) args
       (let* ((context (make-context (if arglist 'struct-boa-constructor 'struct-constructor-name)))
              (name (walk-name walker name context env :def t)))
         (car (push (cons name arglist) (struct-constructors description))))))
    (:copier
     (destructuring-bind (&optional (name (symbolicate "COPY-" struct-name))) args
       (setf (struct-copier-name description)
             (walk-name walker name (make-context 'struct-copier-name) env :def t))))
    (:predicate
     (destructuring-bind (&optional (name (symbolicate struct-name "-P"))) args
       (setf (struct-predicate-name description)
             (walk-name walker name (make-context 'struct-predicate-name) env :def t))))
    (:include
     (destructuring-bind (name &rest slot-descriptions) args
       (setf (struct-include description)
             `(,(walk-name walker name (make-context 'struct-name) env)
               ,@(mapcar (λ (slot-description)
                           (walk-defstruct-slot-description walker slot-description description env))
                         slot-descriptions)))))
    (t (walk-list walker args env))))
```

This code is used in section 342.

344. Explicitly named constructor functions, copiers, and predicates are indexed during the walk of the corresponding options. But if the options are not given, we still need to index the default generated names.

⟨ Index auto-generated structure function names 344 ⟩ ≡

```
(let ((section (nth-value 1 (symbol-provenance name))))
  (flet ((index (name context-type)
           (when name
             (index (walker-index walker) name section (make-context context-type) t))))
    (loop for (name . arglist) in (or (struct-constructors struct-description)
                                      `((,(symbolicate "MAKE-" struct-name))))
          do (index name (if arglist 'struct-boa-constructor 'struct-constructor-name)))
    (index (struct-copier-name struct-description) 'struct-copier-name)
    (index (struct-predicate-name struct-description) 'struct-predicate-name)))
```

This code is used in section 342.

345. Structure slot descriptions are straightforward: they're either a symbol naming the slot, or a list containing the slot name, an optional initform, and possibly the keyword options *:type* and *:read-only*. We don't care about the type, but the *:read-only* option determines whether a reader or an accessor is defined, which we do care about.

⟨ Define *defstruct* parsing routines 342 ⟩ +≡
```
(defun walk-defstruct-slot-description (walker slot-description struct-description env)
  (typecase slot-description
    (symbol
     (walk-defstruct-slot-name walker slot-description nil struct-description env))
    (cons
     (destructuring-bind (slot-name &optional (slot-initform nil initform-supplied) &rest slot-options)
         slot-description
       `(,(walk-defstruct-slot-name walker slot-name slot-options struct-description env)
         ,@(when initform-supplied `(,(walk-form walker slot-initform env)))
         ,@(let ((slot-options (copy-list slot-options)))
             (remf slot-options :type)
             slot-options))))
    (t slot-description)))
```

346. As we walk structure slot names, we'll index the automatically created reader or accessor functions.

In the case of a conflict with an inherited reader or accessor function name, the definition is inherited from the included structure. Because we don't keep structure descriptions around, we can't detect this case directly. But we *can* check to see if there's already a definition in the index for the proposed name, either as a reader or an accessor; if there is, we'll assume it will be inherited and that we should not index it as a definition here.

⟨ Define *defstruct* parsing routines 342 ⟩ +≡
```
(defun walk-defstruct-slot-name (walker slot-name slot-options struct-description env)
  (declare (ignore env))
  (multiple-value-bind (symbol section) (symbol-provenance slot-name)
    (let ((name (symbolicate (struct-conc-name struct-description) symbol))
          (reader (make-context 'struct-slot-reader))
          (accessor (make-context 'struct-slot-accessor))
          (index (walker-index walker)))
      (unless (and (struct-include struct-description)
                   (some #'locator-definition-p
                         (or (find-index-entries index (make-heading name accessor))
                             (find-index-entries index (make-heading name reader)))))
        (index index name section
               (if (getf slot-options :read-only) reader accessor)
               t)))
    symbol))
```

347. ⟨ Define namespace classes 215 ⟩ +≡
```
(defnamespace struct-name () :structure)
(defnamespace struct-constructor-name (function-name) :constructor-function)
(defnamespace struct-boa-constructor (function-name) :boa-constructor)
(defnamespace struct-copier-name (function-name) :copier-function)
(defnamespace struct-predicate-name (function-name) :type-predicate)
(defnamespace struct-slot-reader (function-name) :slot-reader)
(defnamespace struct-slot-accessor (function-name) :slot-accessor)
```

348. Even puns deserve to be properly typeset.

```
(defmethod heading-name :override ((heading struct-boa-constructor))
  "{\\sc␣boa}␣constructor")
```

349. 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)))
```

350. For *defgeneric* forms, we're interested in the name of the generic function being defined, the method combination type, and any methods that may be specified as method descriptions. The environment objects don't keep track of whether a function is generic or not, so we'll have to maintain that information ourselves.

```
(define-special-form-walker defgeneric ((walker indexing-walker) form env &key toplevel)
  (declare (ignore toplevel))
  (destructuring-bind (operator function-name lambda-list &rest options) form
    `(,operator
      ,(note-generic-function
        (walk-name walker function-name
                   (make-context (etypecase function-name
                                   (symbol 'generic-function-name)
                                   (setf-function 'generic-setf-function-name)))
                   env :def t))
      ,(walk-lambda-list walker lambda-list nil env)
      ,@(loop for form in options
              collect (case (car form)
                        (:method-combination ⟨ Walk the method combination option in form 362 ⟩)
                        (:method ⟨ Walk the method description in form 354 ⟩)
                        (t (walk-list walker form env)))))))
```

351. ⟨ Define namespace classes 215 ⟩ +≡

```
(defnamespace generic-function-name (function-name) :generic-function)
(defnamespace generic-setf-function-name (generic-function-name setf-function-name) :generic-setf-function)
```

352. To note that a function is a generic function, we'll stick a property on the function name's plist. (If it's a *setf* function, we'll use the cadr of the function name.) We can then just check for that property, although if the function name is *fboundp*, we'll check the actual type of the function object instead.

```
(defun note-generic-function (function-name)
  (typecase function-name
    (symbol (setf (get function-name 'generic-function) t))
    (setf-function (setf (get (cadr function-name) 'generic-setf-function) t)))
  function-name)
```

```
(defun generic-function-p (function-name)
  (if (fboundp function-name)
      (typep (fdefinition function-name) 'generic-function)
      (typecase function-name
        (symbol (get function-name 'generic-function))
        (setf-function (get (cadr function-name) 'generic-setf-function)))))
```

353. 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
        ,@(when doc `(,doc))
        ,@(when decls `((declare ,@decls)))
        ,@(walk-list walker body-forms env)))))
```

354. ⟨ Walk the method description in *form* 354 ⟩ ≡

```
(let* ((operator (pop form))
       (qualifiers (mapcar (λ (q) (walk-atomic-form walker q nil env))
                           (pop-qualifiers form)))
       (lambda-list (pop form))
       (body form))
  (walk-name walker function-name
             (make-context 'method-name :qualifiers qualifiers)
             env :def t)
  (walk-method-definition walker operator nil qualifiers lambda-list body env))
```

This code is used in section 350.

355. ⟨ Define namespace classes 215 ⟩ +≡

```
(defnamespace method-name (function-name) :method
  ((qualifiers :reader method-qualifier-names :initarg :qualifiers :initform nil)))
(defnamespace setf-method-name (method-name setf-function-name) :setf-method)
```

356. 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 &key toplevel &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 nil env))
                         (pop-qualifiers form)))
     (lambda-list (pop form))
     (body form))
  (declare (ignore toplevel))
  (walk-method-definition walker operator
                          (note-generic-function
                           (walk-name walker function-name
                                      (make-context (etypecase function-name
                                                      (symbol 'method-name)
                                                      (setf-function 'setf-method-name))
                                       :qualifiers qualifiers)
                                      env :def t))
                          qualifiers lambda-list body env))
```

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

```
(defun walk-defclass (walker form context env)
  (destructuring-bind (operator name supers slot-specs &rest options) form
    (throw 'continue-walk
      `(,operator
        ,(walk-name walker name context env :def t)
        ,(mapcar (λ (super) (walk-name walker super context env))
                 supers)
        ,(mapcar (λ (spec) (walk-slot-specifier walker spec env))
                 slot-specs)
        ,@(walk-list walker options env)))))
(define-special-form-walker defclass ((walker indexing-walker) form env &key toplevel)
  (declare (ignore toplevel))
  (walk-defclass walker form (make-context 'class-name%) env))
(define-special-form-walker define-condition ((walker indexing-walker) form env &key toplevel)
  (declare (ignore toplevel))
  (walk-defclass walker form (make-context 'condition-class-name) env))
```

358. 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-name walker spec (make-context 'slot-name) env))
    (cons (destructuring-bind (name &rest options) spec
            `(,(walk-name walker name (make-context 'slot-name) env :def t)
              ,@(loop for (opt-name opt-value) on options by #'cddr
                      if (member opt-name '(:reader :writer :accessor))
                       append `(,opt-name
                                ,(walk-name walker opt-value
                                            (make-context 'method-name :qualifiers (list opt-name))
                                            env :def t))
                      else
                       append `(,opt-name ,opt-value)))))))
```

359. ⟨ Define namespace classes 215 ⟩ +≡

```
(defnamespace slot-name () :slot)
```

360. We'll also walk *define-method-combination* forms to get the names of the method combination types. We'll skip the expansion.

```
(define-special-form-walker define-method-combination ((walker indexing-walker) form env &key toplevel)
  (declare (ignore toplevel))
  `(,(car form)
    ,(walk-name walker (cadr form) (make-context 'method-combination-name) env :def t)
    ,@(walk-list walker (cddr form) env)))
```

361. ⟨ Define namespace classes 215 ⟩ +≡

```
(defnamespace method-combination-name () :method-combination)
```

362. We'll also index custom method combination type uses, which occur in the *:method-combination* option given to a *defgeneric* form.

⟨ Walk the method combination option in *form* 362 ⟩ ≡

```
`(,(car form)
  ,(walk-atomic-form walker (cadr form) (make-context 'method-combination-name) env)
  ,@(walk-list walker (cddr form) env))
```

This code is used in section 350.

363. Indexing for source location tracking. Here is a simple application of the indexing routine: it produces a table describing the locations in the `CLWEB` file at which Lisp names are defined. The table is realized as a simple vector of (*package-name symbol-name namespace* (*lineno*+)) quadruples. The table's contents consist only of strings, keyword symbols, and integers, so that it may be freely dumped and reloaded into a Lisp, no matter what packages exist in that Lisp. This table can be useful for development environments.

```
(defun make-definition-location-table (&optional (sections *sections*) &aux (index (make-index)))
  ; SBCL's authors never heard of the boy who cried wolf.
  (handler-bind (#+sbcl (style-warning #'muffle-warning))
    (index-sections sections :index index))
  (let (results)
    (flet ((extract-entry-info (entry)
             (let* ((heading (entry-heading entry))
                    (symbol (slot-value heading 'name))
                    (sub-heading (sub-heading heading))
                    (locators (entry-locators entry))
                    (definitions (remove-if-not #'locator-definition-p locators)))
               (when (and (typep sub-heading 'namespace) definitions)
                 (push (list (package-name (symbol-package symbol))
                             (symbol-name symbol)
                             (namespace-name sub-heading)
                             (mapcar (λ (locator)
                                       (source-location-lineno
                                        (section-source-location
                                         (elt sections (section-number (location locator))))))
                                     definitions))
                       results)))))
      (map-bst #'extract-entry-info (index-entries index)))
    (coerce (sort results (λ (list1 list2)
                            (and (string≤ (first list1) (first list2))
                                 (string≤ (second list1) (second list2)))))
            'vector)))
```

364. Writing the index. All that remains now is to write the index entries out to the index file. We'll be extra fancy and try to coalesce adjacent locators, so that, e.g., if *foo* is used in sections 1, 2, and 3, the entry will be printed as '*foo*: 1–3'. The function *coalesce-locators* takes a sorted list of locators and returns a list of locators and *section-range* instances. Note that definitional locators will never be part of a range.

```
(defclass section-range ()
  ((start :reader start-section :initarg :start)
   (end :reader end-section :initarg :end)))
(defun coalesce-locators (locators)
  (flet ((maybe-make-section-range (start end)
           (cond ((eql start end) start)
                 ((and start end)
                  (make-instance 'section-range :start (location start) :end (location end))))))
    (do* ((locators locators (cdr locators))
          (loc (car locators) (car locators))
          (coalesced-locators '())
          start end)
         ((endp locators) (nreconc coalesced-locators (ensure-list (maybe-make-section-range start end))))
      (flet ((maybe-push-range (start end)
               (let ((range (maybe-make-section-range start end)))
                 (when range (push range coalesced-locators)))))
        (cond ((locator-definition-p loc)
               (maybe-push-range start end)
               (push loc coalesced-locators)
               (setq start nil end nil))
              ((and end (= (section-number (location loc)) (1+ (section-number (location end)))))
               (setq end loc))
              (t (maybe-push-range start end)
                 (setq start loc end start)))))))
```

365. Here are the pretty-printing routines for the index itself.

```
(set-weave-dispatch 'index
  (λ (stream index)
    (map-bst (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)
            (coalesce-locators
             (sort (copy-list (entry-locators entry)) #'<
                   :key (λ (loc) (section-number (location loc))))))))
(set-weave-dispatch 'section-range
  (λ (stream range)
    (format stream "~D--~D"
            (section-number (start-section range))
            (section-number (end-section range)))))
(set-weave-dispatch 'section-locator
  (λ (stream loc)
    (format stream "~:[~D~;\\[~D]~]"
            (locator-definition-p loc)
            (section-number (location loc)))))
```

366. Entry headings come in many flavors, and need special treatment in order to be printed nicely and correctly. We're intentionally conflating headers with their names here, since the types we accept as names are also valid headings on their own.

The generic function *print-entry-heading* accepts a *&rest* argument only because it's called from a `~/.../` *format* directive; we always ignore the extra arguments.

```
(defgeneric print-entry-heading (stream heading &rest args &key &allow-other-keys))
(defmethod print-entry-heading (stream (heading heading) &key)
  (print-entry-heading stream (slot-value heading 'name))
  (when (sub-heading heading)
    (write-char #\␣ stream)
    (print-entry-heading stream (sub-heading heading))))
(defmethod print-entry-heading (stream (heading namespace) &key)
  (write-string (heading-name heading) stream))
(defmethod print-entry-heading (stream (heading character) &key)
  (print-char stream heading))
(defmethod print-entry-heading (stream (heading string) &key)
  (write-char #\{ stream)
  (print-tex stream (read-tex-from-string heading))
  (write-char #\} stream))
(defmethod print-entry-heading (stream (heading symbol) &key)
  (format stream "\\(~W\\)" heading))
(defmethod print-entry-heading :before (stream (heading pretty-heading) &key)
  (write-string (macro-heading heading) stream))
```

367. Coda: extending the indexer. Here's a self-contained example of how one might extend the indexer to recognize and record specific kinds of forms. In this program, we define reader macro functions for most of the standard macro characters. It might be nice to have dedicated index entries for those definitions that show which macro character is defined where.

368. To begin, we'll define a heading class for macro characters, a subclass for dispatch macro characters, and a constructor function for both. This example is slightly atypical in this regard: most index extensions would simply use a new namespace class and use that as their heading, but we have some specialized sorting requirements that are easiest to fulfill in this way. It's also a bit of a semantic stretch to call macro characters a 'namespace'.

```
(defclass macro-char-heading (heading)
  ((name :reader macro-char :initarg :char)))
(defclass dispatch-macro-char-heading (macro-char-heading)
  ((sub-char :reader macro-sub-char :initarg :sub-char)))
(defmethod sub-heading ((heading dispatch-macro-char-heading))
  (macro-sub-char heading))
(defun make-macro-char-heading (char &optional (sub-char nil sub-char-supplied-p))
  (if sub-char-supplied-p
      (make-instance 'dispatch-macro-char-heading :char char :sub-char (and (characterp sub-char) sub-char))
      (make-instance 'macro-char-heading :char char)))
```

369. When the macro character headings are printed to the index, we'll attach a little label describing what kind of character they are.

```
(defmethod print-entry-heading :after (stream (heading macro-char-heading) &key)
  (format stream "␣~:[~;dispatch␣~]macro␣character"
          (typep heading 'dispatch-macro-char-heading)))
```

370. We'd like the macro character index entries to precede all of the other entries, and we'd like non-dispatching macro chars before dispatching chars. Most types of index entries wouldn't bother with this; they'd just be mixed in with the others in the normal lexicographic ordering.

```
(defmethod entry-heading-lessp ((h1 macro-char-heading) h2)
  (declare (ignore h2))
  t)
(defmethod entry-heading-lessp (h1 (h2 macro-char-heading))
  (declare (ignore h1))
  nil)
(defmethod entry-heading-lessp ((h1 macro-char-heading) (h2 dispatch-macro-char-heading))
  t)
(defmethod entry-heading-lessp ((h1 dispatch-macro-char-heading) (h2 macro-char-heading))
  nil)
(defmethod entry-heading-lessp ((h1 macro-char-heading) (h2 macro-char-heading))
  (char-lessp (macro-char h1) (macro-char h2)))
(defmethod entry-heading-lessp ((h1 dispatch-macro-char-heading) (h2 dispatch-macro-char-heading))
  (and (not (char-lessp (macro-char h2) (macro-char h1)))
       (or (and (not (macro-sub-char h1)) (macro-sub-char h2))
           (and (macro-sub-char h1)
                (macro-sub-char h2)
                (char-lessp (macro-sub-char h1) (macro-sub-char h2))))))
```

371. We also need an equality predicate. We'll say that no macro character heading is equal to anything other than another macro character heading, and that two such headings are equal if their primary and secondary characters are the same (ignoring differences of case).

```
(defmethod entry-heading-equalp ((h1 macro-char-heading) h2)
  (declare (ignore h2))
  nil)
(defmethod entry-heading-equalp (h1 (h2 macro-char-heading))
  (declare (ignore h1))
  nil)
(defmethod entry-heading-equalp ((h1 macro-char-heading) (h2 macro-char-heading))
  (and (char-equal (macro-char h1) (macro-char h2))
       (equalp (sub-heading h1) (sub-heading h2))))
```

372. Next, we'll tell the indexer that the symbols *set-macro-character* and *set-dispatch-macro-character* are 'interesting' and that they should be replaced with referring symbols. (In general, this is dangerous for symbols in the Common Lisp package, so be careful.)

```
(defmethod interesting-symbol-p ((object (eql 'set-macro-character))) t)
(defmethod interesting-symbol-p ((object (eql 'set-dispatch-macro-character))) t)
```

373. And finally, we'll add some methods to *walk-compound-form* specialized on those function names. We're relying here on the around method specialized on *indexing-walker* that swaps out referring symbols for the *operator* argument but leaves them as the car of the *form*. Also notice that by overriding the default method and not calling *index-name*, we can prevent the functions themselves from being indexed, which is what we want in this case.

```
(defmethod walk-compound-form ((walker indexing-walker) (operator (eql 'set-macro-character))
                               form env &key toplevel)
  (declare (ignore toplevel))
  (multiple-value-bind (symbol section) (symbol-provenance (car form))
    (when (and section (characterp (second form)))
      (add-index-entry (walker-index walker)
                       (make-macro-char-heading (second form))
                       section t))
    `(,symbol ,@(walk-list walker (cdr form) env))))
(defmethod walk-compound-form ((walker indexing-walker) (operator (eql 'set-dispatch-macro-character))
                               form env &key toplevel)
  (declare (ignore toplevel))
  (multiple-value-bind (symbol section) (symbol-provenance (car form))
    (when (and section (characterp (second form)))
      (add-index-entry (walker-index walker)
                       (make-macro-char-heading (second form) (third form))
                       section t))
    `(,symbol ,@(walk-list walker (cdr form) env))))
```

374. Index.

⟨ Accumulate TEX-mode material in *commentary* 135 ⟩ Used in section 132.
⟨ Accumulate limbo text in *commentary* 133 ⟩ Used in section 132.
⟨ Accumulate Lisp-mode material in *code* 136 ⟩ Used in section 132. Cited in section 139.
⟨ Build a logical block from *list* 187 ⟩ Used in section 186.
⟨ Call the standard reader macro function for #⟨*sub-char*⟩ 104 ⟩ Used in section 103.
⟨ Check for an ambiguous match, and raise an error in that case 41 ⟩ Used in section 39.
⟨ Check for *&whole* and *&environment* vars and augment the environment if found 262 ⟩ Used in section 261.
⟨ Clean up after indexing 317 ⟩ Used in section 323.
⟨ Compilation unit for tangling 151 ⟩ Used in section 150.
⟨ Complain about any unused named sections 152 ⟩ Used in section 150.
⟨ Complain about starting a section without a commentary part 137 ⟩ Used in section 136.
⟨ Condition classes 36, 40, 53, 126, 138, 153 ⟩ Used in section 5.
⟨ Construct a declaration specifier for special variables in *names* 260 ⟩ Used in section 258.
⟨ Copy any macro function associated with *symbol* to *refsym* 313 ⟩ Used in section 312.
⟨ Declare *structure-description* structure 341 ⟩ Used in section 340.
⟨ Define namespace classes 215, 216, 217, 347, 351, 355, 359, 361 ⟩ Used in section 214.
⟨ Define the pprint dispatch table setters 148, 165 ⟩ Used in section 74.
⟨ Define *defstruct* parsing routines 342, 345, 346 ⟩ Used in section 340.
⟨ Define *heading-name* generic function 294 ⟩ Used in section 290.
⟨ Ensure that every referenced named section is defined 154 ⟩ Used in section 150.
⟨ Extract the required parameters from *lambda-list* 270 ⟩ Used in section 269.
⟨ Find declarations in *declare* that apply to variables in *names* 259 ⟩ Used in section 258.
⟨ Find the tail of the list marker 80 ⟩ Used in section 79.
⟨ Finish the last section and initialize section variables 134 ⟩ Used in sections 132 and 137.
⟨ Fix up the index and sections filenames for test suite output 159 ⟩ Used in section 157.
⟨ Global variables 12, 17, 24, 27, 42, 46, 49, 50, 55, 62, 65, 76, 145, 146, 147, 158, 160, 164, 183, 287, 303, 308, 315, 322 ⟩ Used in section 5.
⟨ Index an expanded form 327 ⟩ Used in section 326.
⟨ Index auto-generated structure function names 344 ⟩ Used in section 342.
⟨ Initialize global variables 25, 28, 43, 288, 304, 316 ⟩ Used in sections 143, 150, and 157.
⟨ Insert a new node with key *item* and return it 33 ⟩ Used in section 32.
⟨ Make *refsym* an alias for any class named by *symbol* 314 ⟩ Used in section 312.
⟨ Maybe print a package prefix for *symbol* 180 ⟩ Used in section 179.
⟨ Output a form that sets the source pathname 156 ⟩ Used in section 155.
⟨ Print *symbol-name* 181 ⟩ Used in section 179.
⟨ Process the list *x* for backquotes and commas 94 ⟩ Used in section 93.
⟨ Process the symbol in *arg* as a parameter 268 ⟩ Used in sections 264, 265, 266, and 267.
⟨ Process *arg* as a keyword parameter 266 ⟩ Used in section 261.
⟨ Process *arg* as a required parameter 264 ⟩ Used in section 261.
⟨ Process *arg* as an auxiliary variable 267 ⟩ Used in section 261.
⟨ Process *arg* as an environment parameter 263 ⟩ Used in section 261.
⟨ Process *arg* as an optional parameter 265 ⟩ Used in section 261.
⟨ Read '0's and '1's from *stream* 101 ⟩ Used in section 100.
⟨ Read characters up to, but not including, the next newline 88 ⟩ Used in section 87.
⟨ Read the next object from *stream* and push it onto *list* 84 ⟩ Used in sections 82 and 83.
⟨ Read the next token from *stream*, which might be a consing dot 83 ⟩ Used in section 82.
⟨ Should we include the last character of *raw-input*? 72 ⟩ Used in section 71.
⟨ Signal an error about section definition in Lisp mode 127 ⟩ Used in section 125.
⟨ Signal an error about section name use in TEX mode 128 ⟩ Used in section 125.
⟨ Split *form* into lines, eliding any initial indentation 205 ⟩ Used in section 204.
⟨ Split *symbol-name* into a prefix and nicely formatted suffix 182 ⟩ Used in section 181.
⟨ Trim whitespace and reverse *commentary* and *code* 139 ⟩ Used in section 132.

⟨ Update the section name if the new one is better 48 ⟩ Used in section 47.
⟨ Walk one *defstruct* option 343 ⟩ Used in section 342.
⟨ Walk the 'ignore' or 'ignorable' declarations in *data* 256 ⟩ Used in section 254.
⟨ Walk the 'special' declarations in *data* 255 ⟩ Used in section 254.
⟨ Walk the atom *form* 228 ⟩ Used in section 227.
⟨ Walk the method combination option in *form* 362 ⟩ Used in section 350.
⟨ Walk the method description in *form* 354 ⟩ Used in section 350.
⟨ Walk *form* 227 ⟩ Used in section 225.
⟨ Walker generic functions 224, 226, 230, 232, 237, 239, 257 ⟩ Used in section 222.
⟨ Weave the sections to the output file, reporting as we go 162 ⟩ Used in section 161.
⟨ Write the program name to the tests output file 163 ⟩ Used in section 161.
⟨ *:if-exists* default disposition 149 ⟩ Used in sections 150 and 161.

CLWEB