Literate Programming in Lisp (LP/Lisp)

Roy M. Turner

Technical Report 2010–2
Department of Computer Science
University of Maine
Orono, ME 04469

Contents

1 Introduction

This document describes LP/Lisp (Literate Programming/Lisp), a Lisp program that supports literate Lisp programming.

Writing a program and writing its documentation generally are considered two separate tasks. This leads to at least two problems: first, the documentation is often never written; and second, when it is written, it is often an afterthought. Literate programming, introduced and with early development by Donald Knuth, basically consists of writing programs that can be considered works of literature. The program and its documentation are written together, with the goal of informing not only the computer, but also the reader. Literate programming tools then usually take the combined file and split it out for the purpose at hand: a source file for compiling or a file for typesetting.

You may ask, why another literate programming tool? Why not use something like the excellent `noweb`[1] package? Well, there are some problems with using that model of literate programming for interpreted languages like Lisp. For a compiled language, introducing another step—translating from the literate programming "source" file to the language's source code—is not particularly problematic. However, the typical Lisp programming model involves incremental development and, most of all, interactive debugging. So the programmer usually writes the code, then, when there is a bug, fixes the problem in an Emacs buffer,[2] evaluates the changed definition, and continues. With something like `noweb`, this isn't really practical. Changes to in the Lisp code don't migrate back to the `noweb` source code, and there are no tools I'm aware of that support debugging and incremental evaluation directly from the `noweb` source.

LP/Lisp takes a different approach, one more appropriate for Lisp: documentation is embedded in the Lisp file, not a separate file written in literate programming markup, and the LaTeX documentation is produced from that via this Lisp program. The literate programming file *is* the Lisp source code, and no other step stands between the programmer and Lisp.

We do lose some things, however. For example, unlike `noweb`, LP/Lisp does not support splitting chunks of Lisp code and then reassembling them, for obvious reasons, although as you will see, you can insert LaTeX source in the middle of Lisp functions as comments. And you can reorder whole chunks of the source file in the typeset output, allowing such things as introductory sections of the file to be squirreled away at (e.g.) the end of the file and put at the right place for typesetting.

LP/Lisp is tied to two pieces of software that limit its portability. First, it only produces LaTeX output. This, to me, is not a limitation, as this is the best typesetting tool available. Second, however, LP/Lisp is written in and has been tested in only one version of Lisp, Allegro Common Lisp (ACL)[3]. It is unlikely to work in other versions of Lisp, since it relies on a regular expression package provided in that Lisp. However, since ACL is freely-available as a trial version, you should be able to run LP/Lisp even if you do not use ACL on a daily basis yourself.

2 Documentation in a Lisp File

The Lisp file's is processed by LP/Lisp to create the LaTeX source. Comments in the Lisp file contain the LaTeX text, while the Lisp code itself is in-lined and typeset as code.

There are two documentation styles supported by LP/Lisp, implicit and explicit.

[1] http://www.cs.tufts.edu/ nr/noweb

[2] Sure there are other development environments, but not for *real* Lisp programmers, right?

[3] http://www.franz.com

2.1 Implicit Documentation

In the implicit documentation style, LaTeX source code is generated from both the Lisp comments and the Lisp code itself. Three kinds of comments are possible in Lisp: long comments, short (i.e., single-line) comments, and comments appearing on the same line as Lisp code. Except for embedded commands and markup (see below), LP/Lisp pulls text out of the first two kinds of comment intact as LaTeX source.

Long comments are defined using Lisp's block comment style:

```
#|
This is a block comment.

Anything is allowed between the delimiters.
|#
```

Shorter comments have the form of a line in which the first non-whitespace character is a semicolon. Everything following the semicolons are treated as LaTeX code, as well.

Note that it is possible to have LP/Lisp ignore some comments by placing a percent sign % as the first non-whitespace character after the leading semicolons. This is sometimes useful for comments that are useful for debugging, etc., but that would not make much sense to the reader of the LaTeX version.

In addition, LP/Lisp can be made to ignore blocks of code and LaTeX by using its own `comment` embedded command (see below).

2.2 Explicit Documentation

In the explicit documentation style, LaTeX source code is still generated from Lisp code and comments, but now only a subset of the comments are converted to LaTeX, and some may be omitted entirely from the LaTeX file.

In this style, the user has control of which comments will be converted to LaTeX and which will be typeset as Lisp code. This is specified using XML-like tags saying where the documentation begins and ends.

To specify that a long comment should be typeset as LaTeX, it should look like:

```
#|<doc>
 Your \LaTeX source goes here.
</doc>
|#
```

This would become:

> Your LaTeX source goes here.

in the finished product.

The <doc> tag, which must be positioned as shown,[4] tells LP/Lisp that the LaTeX code is starting, and </doc> tells it the code is done. LP/Lisp is pretty forgiving about ending documentation tags: if </doc> is omitted, then the LaTeX ends at |#.

Short comments can also specify LaTeX source code. This is done in pretty much the same way:

[4] For the moment, anyway.

```
;;;<doc> Your \LaTeX source goes here.
;;;</doc>
```

which would produce the same thing. As with long comments, LP/Lisp is forgiving about the end: the LaTeX is assumed to be concluded when we encounter something that is not a short comment.

For the short comment form, a there is a shorthand form as well to avoid cluttering the Lisp code for short LaTeX comments:

```
;;;[ Your \LaTeX source goes here.
;;;]
```

will also work. Note, however, that this:

```
;;;[Your \LaTeX source goes here.]
```

will likely not do what you intended: the closing square bracket will appear in your LaTeX file.[5] However,

```
;;;[Your \LaTeX source goes here.
```

will produce what you want, as long as the next line is not a short comment.

LP/Lisp allows aliases for the "doc" tag for the user's convenience. These are:

- `<d>...</d>`
- `<latex>...</latex>`
- `<l>...</l>`

You should also be aware that in the explicit documentation mode, although most non-marked comments appear typeset as Lisp code, not all do. In particular, any comment that begins with a semicolon in column 1 is omitted. This is done to allow you to comment out sections of code, to have Lisp headers that do not appear in the LaTeX documentation, and so forth.

2.3 Embedded LP/Lisp Markup

LP/Lisp defines some markup conventions to make it both easier to specify LaTeX commands as well as to make the Lisp comments cleaner. Marked-up text is surrounded by double square brackets ([[`text`]]), much as in `noweb`. There are two versions of this. If the leading brackets are immediately followed by one or more asterisks (*), then the text will be the argument to a LaTeX sectioning command. (The same number of asterisks should precede the closing brackets, as well.) The number of asterisks determine the level of section: one means section, two means subsection, three means subsubsection, four means paragraph, and five means subparagraph. For example,

```
[[* Introduction *]]
[[** Markup **]]
```

would become:

```
\section{Introduction}
\subsection{Markup}
```

[5] This should be fixed in a later version.

in the LaTeX source file.

One special type of markup that looks something like an embedded command (see below) is the following:
<*definition-type name*>

This is used to specially set off definitions in the LaTeX code (and, incidentally, in the Lisp comments, too). Here, *definition-type* is the kind of thing being defined, one of: function, macro, class, method, variable, parameter, or generic. The *name* is the name of the thing being defined. The result will be a new boldfaced heading in the LaTeX output.

As an example of using this, consider the following:

```
;;;<function foobar>
```

which would produce in the printed output:

Function: foobar

2.4 Embedded LP/Lisp Commands

In addition to the documentation tags, embedded LP/Lisp commands are allowed in the Lisp file. These can be used to specify some LP/Lisp parameters (which can also be specified in the call to `make-latex`), to define chunks of text and to insert chunks in-line, to include LaTeX code in the preamble, and to ignore or comment out sections of the file to prevent LP/Lisp processing (but not Lisp processing!).

All embedded commands must, of course, be ignorable by Lisp, so they have to be inside of comments. On the other hand, LP/Lisp has to be able to see them and recognize them, so they should be distinct from most LaTeX commands.

We use a simple version of XML-like tags for this, although all within Lisp comments. For example, to define the document's title, you would do something like:

```
;;; <title>This is the Document Title</title>
```

This could span multiple (comment) lines, as well. Valid things to set this way include those parameters also specifiable via the command line:

<`title`>**text**</`title`> – Define the document title. This can be overridden by the corresponding `:title` parameter of

- <`title`>text</`title`> – Define the document title.
- <`author`>text</`author`> – Define the document's author.
- <`date`>text</`date`> – Define the document's date.
- <`options`>text</`options`> – Define `documentclass` options. This should be a comma-separated list (LaTeX, not Lisp!) of options, e.g., `12pt,landscape`.
- <`packages`>text</`packages`> – Define any additional packages to be loaded by LaTeX. This is a comma-separated list of package names, appropriate for the argument to `usepackage`. If you need anything more fancy than that, use `preamble`.

- <preamble>text</preamble> – Define LaTeX commands to add to the document's preamble (i.e., between the documentclass and begin{document} commands).
- <comment>text</comment>, <ignore>text</ignore> – LP/Lisp ignores everything within these. Note that Lisp does not!
- <toc/> – If specified, include a table of contents.
- <complete/> – If specified, write out the complete LaTeX file, ready for processing by latex. If not, then write only the portion that would come between the begin{document} and end{document} commands.
- <define name="name">text</define> – Define a chunk of text and call it by the name name. The defined text is not output as LaTeX unless it is named in an insert embedded command.
- <insert name="name"/> – Insert the defined chunk of text named name here in theLaTeX file.
- <chunk name="name">text</chunk> – Define a chunk of code and call it by the name name. A tag is inserted at the point where the chunk was defined. It can later be inserted using <insert-chunk>.
- <insert-chunk name="name"/> – Insert the defined chunk of code named name here in theLaTeX file.
- <implicit> or <explicit> – Specify whether the implicit or the explicit documentation style is to be used.

To use the define embedded command, make sure that the command is within a comment and that you are clear about the comment status of what is being defined for later inclusion. For example, although it is tempting to do this:

```
#|
<define name="introduction">
  ...
</define>
|#
```

It may not work as you expect. This is because more than likely, the insert command will have been specified as something like:

```
;;;<insert name="introduction"/>
```

This will result in the insert command being replaced by the defined lines, but *without the long comment start and end characters*! What works better is either to define the block as:

```
;;; <define name="introduction">
#|
  ...
|#
;;; </define>
```

or else to wrap the `insert` in a long comment, i.e.:

```
#|
<insert name="introduction"/>
|#
```

The <chunk> command is used so that portions of your Lisp code can be elided and later referred to where it may make more sense to talk about it, in the manner that other literate programming tools, such as `noweb`, allow. For example, suppose we had the following:

```
(defun factorial (n)
  (cond
    ;;<chunk name="base-case">
    ((<= n 1) 1)
    ;;</chunk>
    ;;<chunk name="recursive-step">
    (t
      (* n (factorial (- n 1))))
    ;;</chunk>
   )
 )
```

This would, in the output, look something like:

```
(defun factorial (n)
  (cond
    <<base-case>>
    <<recursive-step>>
   )
 )
```

We could then use <insert-chunk> to talk about the chunks. For example,

```
The recursive step for factorial looks like:
<insert-chunk name="recursive-step">
which is pretty simple.
```

This would look in the output something like:

```
The recursive step for factorial looks like:
   [00030] <<recursive-step>>=
   [00031]     (t
   [00032]      (* n (factorial (- n 1))))
```

Note that you cannot at this time embed chunks within other chunks, nor can you embed them within other defined things! (Not and have it work correctly, in any case.)

2.5 Lisp Code Formatting

LP/Lisp outputs Lisp code prefaced by line numbers. These line numbers do not correspond to where the Lisp code lives in the actual `.lisp` file, but rather are reference numbers for the reader for the non-commented Lisp code. We assume that your IDE will help you find the appropriate Lisp line during debugging without needing source file line numbers.

2.6 Automatic Indexing

LP/Lisp creates index entries for each thing it can recognize as being defined, such as functions (by `defun`), classes (`defclass`), and so forth. These will cause entries to be made in the file's corresponding `.idx` file, as usual for LaTeX. The output `.tex` file will also, if the `complete` parameter is set, have code at the end that loads an index (`.ind`) file that can be produced by (e.g.) the `makeindex` command. LP/Lisp also "touches" this file, which creates if it doesn't exist, so that even if you haven't yet run `makeindex`, you can still run `latex` on the file.[6]

3 Using LP/Lisp

You can use LP/Lisp either from within Lisp or from the shell's command prompt.

To use LP/Lisp from within Lisp, you (obviously) have to load it first. If the directory in which LPLisp resides has been added to the `sys:*require-search-list*` variable,[7], you should be able to load LP/Lisp simply by typing:

```
(require :lplisp)
```

If not, then you will need to find out where the file `lplisp.fasl` was installed and load it using `load`.

Once loaded, you can simply call the `make-latex` function. This function is in the `lplisp` package, however, so you will need to call it as:

```
(lplisp:make-latex lisp-file keyword-options)
```

where *lisp-file* is your Lisp source file.

`make-latex` has many keyword arguments, as shown in Figure 1. Note that most of these only make sense if `complete?` is specified. For example, if `complete?` is not specified, no `title` will be output.

LP/Lisp can also be run from a shell using the `lplisp` command. This command is actually a Lisp program that is called using the standard shell script starting conventions.[8] In fact, the function `write-shell-script` creates this file as part of the installation process of LP/Lisp. (See Section 4.)

The script takes the same basic arguments as the Lisp function `make-latex` described above. The first argument should be the Lisp file to be processed. The other arguments are shown in Figure 2.

3.1 Running LP/Lisp on Multiple Files

Sometimes it is convenient to run LP/Lisp on a set of files that together comprise a program. In this case, you will likely want one complete LaTeX file and others that can be included or input into it.

For example, suppose that your program consists of the files `robot.lisp`, `agent.lisp`, and `hardware.lisp`, with `robot.lisp` being in some sense the main file and the others holding support

[6]Due to the vagaries of LaTeX and LP/Lisp itself, there may be a noticeable problem with spacing just after definitions; this is due to the way the `index` command is inserted. This should be fixed in the future.

[7]If you are reading this at MaineSAIL, it already has been.

[8]I.e., the first line starts with #!, sometimes called "shebang", and the name of the Lisp executable, which causes the shell to start Lisp and pass the file to it.

Keyword	Argument type	Description
`complete?`	Boolean	Output a complete LaTeX file.
`toc?`	Boolean	Include a table of contents.
`packages`	list of strings	Use these packages as well.
`options`	list of strings	Options to add to the `documentstyle` command.
`latex-file`	string	LaTeX file to output – if not specified, then the base name of the positional argument is used as the base for the output file, with ".tex" as its file extension. If this is a directory, then the base filename is used for the Lisp file is used to create a filename in that directory for the .tex file. Finally, if there are multiple Lisp files to process, this is a directory, and `output-directory` is unspecified, then the output files are put in that directory.
`output-directory`	string	This specifies a directory that output files should be placed in. For single files, this is combined with either the value of `latex-file` or the base filename of the Lisp input file. For multiple input files, then the value of `latex-file` is ignored, and this is combined with the base filename of the input files.
`title`	string	The title of the file; overrides the corresponding embedded command.
`author`	string	The author of the file; overrides the corresponding embedded command.
`date`	string	The date of the file; overrides the corresponding embedded command.
`documentation-type`	keyword	Either `:implicit` or `:explicit`.

Figure 1: Keyword parameters of the `make-latex` function.

functions. In this case, you would likely the skeleton of the documentation in `robot.lisp` and use LaTeX's `include` or `input` facility to include the LaTeX files produced from the other two:

```
---in robot.lisp---
;;;\include{agent}
;;;\include{hardware}
```

You could run LP/Lisp on each of these separately, specifying for each whether you want a complete LaTeX file or not. However, LP/Lisp allows you to do this more simply by specifying a list of files to `make-latex` or multiple files on the `lplisp` command line:

```
(make-latex '("robot.lisp" "agent.lisp" "hardware.lisp") ...)
```

```
lplisp {options} robot.lisp agent.lisp hardware.lisp
```

In either case, the setting of the `complete?` (or `-c`, or `--complete`) parameter will only apply to the first file. The others will have that parameter set to nil.

Argument	Value	Description
`-c, --complete`	none	Output a complete LaTeX file.
`--toc`	none	Include a table of contents.
`-p, --package`	string	Specify a LaTeX package or package to use in addition to any specified in the file's <`package...`> embedded command. To specify more than one package, either use multiple `-p` or `--package` arguments or specify them separated by commas, e.g.: [[-p cdps,html]].
`-o, --option`	string	Options to use in the `documentstyle` command, in addition to any specified by the <`options...`> embedded command. To specify multiple options, either use multiple `-o` or `--option` arguments, or specify them separated by commas, e.g.: [[-o 12pt,landscape]].
`-l, --latex`	string	LaTeX file to output – if not specified, then the base name of the positional argument is used as the base for the output file, with ".tex" as its file extension. If this is a directory, then the base filename is used, with the .tex extension, for an output file within that directory. If there are multiple files and `-f` is not set, then the files are placed in this directory.
`-f, --ldir`	directory	The output directory for the `.tex` files produced by `lplisp`. For single input files, this is combined with either the value of the `-l` option to form an output filename, or else it is combined with the base filename of the Lisp file. For multiple files as input, then this is used with the base filenames of the Lisp files, and the `-l` option is ignored.
`-t, --title`	string	The title of the file; overrides the corresponding embedded command.
`-a, --author`	string	The author of the file; overrides the corresponding embedded command.
`-d, --date`	string	The date of the file; overrides the corresponding embedded command.
`-e, --explicit`	none	Use the explicit documentation style.
`-i, --implicit`	none	Use the implicit documentation style.

Figure 2: Arguments of the `lplisp` shell command.

4 Installing LP/Lisp

The LP/Lisp distribution consists of two files, the source file (`lplisp.lisp`) and the shell script (`lplisp`). The shell script is not actually necessary, as LP/Lisp can create it, as we will see.

To install LP/Lisp, compile the source file from inside of Lisp using `compile-file`, then put the resulting `.fasl` file where it can be easily loaded at your site. The best thing to do is to put it where Lisp's `require` function can find it, that is, in a directory on the `sys:*require-search-list*` list.

You can either edit the shell script directly or generate a new one. If you edit it directly, you will need replace file name on the first line with the name of your Lisp executable. You will also need to change the directory from which the `lplisp.fasl` file is loaded, on the line beginning `(load...)`.

To generate a new version of the shell script, load LP/Lisp, then use the `write-shell-script` function. This function by default will create a new shell script named `lplisp` in Lisp's current directory. This is likely not what you wanted. You can either use the `:cd` command to change the

working directory before calling the function, or you can specify where it goes using the function's `LPLisp-directory` keyword argument. You can also specify a Lisp image name using the keyword argument `image-name`; if you do not, the image that you are running at that point will be used.

Regardless of which method you use, you will need to `chmod` the shell script appropriately and put it somewhere on your shell load path.

5 Known Issues

Given the variety of code and comments that need to be addressed, there are likely to be remaining formatting issues that need to be fixed. This section lists those that are currently known.

5.1 Text that looks like LP/Lisp commands confuses LP/Lisp

You should not use things in your documentation, even within verbatim mode, that look like LP/Lisp embedded commands, as the program will issue an error message about nested commands. If you need to have something like <`thing`> in your documentation, you need to generate this some other way than literally. This is a pain for verbatim, I know (trust me, I know). One possibility is to use the `alltt` package instead of verbatim; another is to split the verbatim into pieces, for example, by using multiple `verb` commands.

5.2 Spurious long comment characters in explicit mode

At this time, there seems to be no clean way to get rid of the long comment begin/end characters when in explicit mode and the long comment is used to define a `define` embedded command. The definition is taken out, but the characters remain. This should not happen in implicit mode.

6 Change History

21:48 - Thu Mar 5, 2009 -rmt- Began to change to allow alternate way to specify which comments to extract.

7 The Program

The program makes two passes through your Lisp code. First, it reads in each line and, instead of outputting it to the final file, sticks it into the `*lines*` list. If we encounter embedded commands, they go in the `*embedded-commands*` association list (alist), with their name as the key. (Two exceptions: the `define` command, which has both `define` and the name of the chunk being defined as the key; and `insert`, which is left in `*lines*`.) Once the file has been read, we insert any chunks defined by `define` commands where their corresponding `insert` commands were embedded. We then go through `*lines*` again, this time outputting LaTeX code based on what we've seen. The bulk of this happens in the `process-lines` function.

7.1 Getting Started

7.1.1 Lisp Utilities

LP/Lisp requires a couple of things. First, we need the perl-like regular expression package `regexp2`:

```
[00001] (require 'regexp2)
[00002]
```

and also our local utilities, which at this time are loaded not by `require`, but rather by old-fashioned package loading.[9]

```
[00003] #-:MaineSAIL-utilities
[00004] (load "util:;load")
[00005]
```

7.1.2 Variables

Next, we need to declare some variables to hold the lines of the file (`*lines*`) and to hold the various embedded commands (`*embedded*`).

```
[00006] (defvar *lines* nil)
[00007] (defvar *embedded-commands* nil)
[00008]
[00009]
```

We need to keep track of what line we're on, too, so we can tell the user when there's a problem. We also need a line number counter for the Lisp code (minus the comments), `*codeLineNo*`.

```
[00010] (defvar *lineno* 0)
[00011] (defvar *codeLineNo* 0)
[00012]
[00013]
```

We also need to define the valid embedded commands. The variable `*valid-embedded-commands*` contains these. Each entry has one of these two forms:

```
(command-name . function)
((command-name alternate-name...) . function)
```

The command names (and alternate names) should be strings. Note that some of these have as their function `embedded-unexpected-tag`; this is because these tags should have been found by other functions.

```
[00014] (defparameter *valid-embedded-commands*
[00015]     '(
[00016]       ("insert" . embedded-insert)
[00017]       ("/insert" . embedded-unexpected-tag)
[00018]       ("define" . embedded-define)
[00019]       ("/define" . embedded-unexpected-tag)
[00020]       ("chunk" . embedded-chunk)
[00021]       ("/chunk" . embedded-unexpected-tag)
[00022]       ("insert-chunk" . embedded-insert-chunk)
[00023]       ("/insert-chunk" . embedded-unexpected-tag)
[00024]       ("toc" . embedded-toc)
[00025]       ("complete" . embedded-complete)
[00026]       ("title" . embedded-title)
```

[9] Note that we only use a few things out of this, most notably the message output functions; these could easily be included in (or rewritten for) a standalone distribution.

```
[00027]       ("/title" . embedded-unexpected-tag)
[00028]       ("author" . embedded-author)
[00029]       ("/author" . embedded-unexpected-tag)
[00030]       ("date" . embedded-date)
[00031]       ("/date" . embedded-unexpected-tag)
[00032]       ("packages" . embedded-packages)
[00033]       ("/packages" . embedded-unexpected-tag)
[00034]       ("options" . embedded-options)
[00035]       ("/options" . embedded-unexpected-tag)
[00036]       ("preamble" . embedded-preamble)
[00037]       ("/preamble" . embedded-unexpected-tag)
[00038]       ("comment" . embedded-comment)
[00039]       ("/comment" . embedded-unexpected-tag)
[00040]       ("ignore" . embedded-ignore)
[00041]       ("/ignore" . embedded-unexpected-tag)
[00042]       ("function" . embedded-function)
[00043]       ("functions" . embedded-function)
[00044]       ("method" . embedded-function)
[00045]       ("methods" . embedded-function)
[00046]       ("class" . embedded-function)
[00047]       ("classes" . embedded-function)
[00048]       ("generic" . embedded-function)
[00049]       ("generics" . embedded-function)
[00050]       ("variable" . embedded-function)
[00051]       ("variables" . embedded-function)
[00052]       ("parameter" . embedded-function)
[00053]       ("parameters" . embedded-function)
[00054]       ("macro" . embedded-function)
[00055]       ("macros" . embedded-function)
[00056]       ("implicit" . embedded-implicit-documentation-function)
[00057]       ("explicit" . embedded-explicit-documentation-function)
```

Explicit documentation tags: since we don't know at the time this is called (from within `read-input`) whether or not `*implicit-documentation*` is set – it could be set by a future embedded command – we have to punt on processing these. Consequently, we just allow the line to be put on `*lines*` as is.

```
[00058]       (("latex" "l" "doc" "d" "/latex" "/l" "/doc" "/d") . embedded-nop-function)
[00059]       ("break-read" . embedded-break-read)
[00060]       ("break" . embedded-break)
[00061]       ("debug" . embedded-debug)
[00062]       ("debug-off" . embedded-debug-off)
[00063]       )
[00064]   )
[00065]
```

Variable *implicit-documentation*

This variable controls which comments LP/Lisp converts into LaTeX code. If `*implicit-documentation*` is set, then all comments are processed into LaTeX code. If it is nil, then only those that are explicitly marked (see Section 3) are converted, and other Lisp comments are included in the Lisp code listing.

```
[00066] (defvar *implicit-documentation* t)
[00067]
```

Parameter *allow-embedded-latex*

This parameter controls whether or not you can have LaTeX commands within Lisp code. The default is to allow them, but sometimes, you'll notice that when you run LaTeX on your file, you'll get cryptic messages from some of the Lisp lines. (My current favorite is "Missing number, treated as zero.", flagging the `\unhbox` command – that isn't even in the Lisp code!) This is often due not to LaTeX commands you may have embedded (say, in in-line comments), but rather to things like strings and regular expressions in your code. If you set `*allow-embedded-latex*` to `nil`, this should stop.

```
[00068] (defparameter *allow-embedded-latex* t)
[00069]
[00070]
```

Variable *processing-explicit-documentation*

This global variable is used to track whether we're in the middle of processing a piece of explicit documentation or not.

```
[00071] (defvar *processing-explicit-documentation* nil)
[00072]
```

Parameter *documentation-tags*

Valid explicit documentation tags.

```
[00073] (defparameter *documentation-tags*
[00074]     '("latex" "l" "doc" "d"))
[00075]
```

The `*standard-preamble*` variable contains any preamble commands, as a list of strings, that should be included in the preamble of the LaTeX output file. For example, if there are any LP/Lisp-associated style files, this is the place to put `usepackage` commands for them. Keep in mind that you need to escape any backslashes!

```
[00076] (defparameter *standard-preamble*
[00077]     '(
```

Use the `alltt` package for pseudo-verbatim:

```
[00078]       "\\usepackage{alltt,makeidx}"
[00079]       "\\def\\lplispHyphen{-}"
[00080]       "\\def\\lplispRArrow{$>$}"
[00081]       ))
[00082]
[00083]
```

For debugging, we allow single-stepping during the processing of the input lines from `*lines*`. To turn this on, call `debug-lplisp` with no arguments; to turn it off, call it with nil as its argument. The global variable used is `*debug-lplisp*`.

```
[00084] (defvar *debug-lplisp* nil)
[00085]
```

The parameter `*tab-width*` let's LP/Lisp know how many spaces to insert for tabs in Lisp code. It defaults to Emacs' default of 8 spaces/tab.

```
[00086] (defparameter *tab-width* 8)
[00087]
```

This parameter, `*no-lisp-yet*` keeps track of whether or not the first line of non-blank Lisp code has been written; this keeps spurious lines of blanks being printed at the top of the LaTeX file.

```
[00088] (defparameter *no-lisp-yet* t)
[00089]
```

This parameter, `*defines-contain-nested-commands*`, is set by the code that handles definitions (<`define...`>), if one or more of the defines contains nested commands. If so, then we have to do an additional step, and a costly one: write everyting to an intermediate temp file, then read it back in.

```
[00090] (defvar *defines-contain-nested-commands* nil)
[00091]
```

7.2 The `make-latex` Function

The `make-latex` function is the "entry point" for LP/Lisp. It takes one positional argument (the Lisp file(s) on which to run LP/Lisp) and several keyword arguments, as described in Section 3. Basically, this calls functions to read the input file into `*lines*` (while collecting embedded commands), then insert the embedded commands into the `*lines*`, then processing the lines and outputting them.

```
[00092] (defun make-latex (lisp-file &key (complete? t)
[00093]                                   (toc? t)
[00094]                                   packages
[00095]                                   (options nil options-set?)
[00096]                                   latex-file
[00097]                                   (title nil title-set?)
[00098]                                   (author nil author-set?)
[00099]                                   (date nil date-set?)
[00100]                                   (documentation :implicit)
[00101]                                   (allow-embedded-latex t)
[00102]                                   output-directory
[00103]                                   )
```

First set the initial setting of the documentation, either implicit or explicit:

```
[00104]   (setq *implicit-documentation*
[00105]     (if (eql documentation :explicit)
[00106]         nil
[00107]       t))
[00108]
[00109]   (setq *allow-embedded-latex* allow-embedded-latex)
[00110]
[00111]   (cond
```

If the `lisp-file` argument is a list of files, then we need to recursviely call `make-latex` on each one in turn. The problem is, what to do with `complete?` or embedded `complete` commands? We take the approach of using the parameter `complete?` as set for the first one, and nil for the rest. Note that we use a rather roundabout way of composing the recursive call, including using `eval`, in order to preserve the meaning of unspecified keyword parameters in the original call:

```
[00112]    ((listp lisp-file)
[00113]     (cond
[00114]      (output-directory
[00115]       (if (probe-directory output-directory)
[00116]           (setq latex-file nil)
[00117]         (error "Output directory ~s is not a directory!~%"
[00118]                output-directory)))
[00119]      ((null latex-file)
[00120]       (format t "Writing .tex files in current directory.~%"))
[00121]      ((probe-directory latex-file)
[00122]       (format t "Using ~a as output directory.~%" latex-file)
[00123]       (setq output-directory latex-file)
[00124]       (setq latex-file nil))
[00125]      (t
```

So there's no output directory specified, but there is a non-directory `latex-file` specified – what could the user mean?

```
[00126]       (error
[00127]        "No output directory specified for multiple files, yet a LaTeX output file was specified
[00128]       ))
[00129]
[00130]     (loop with first? = t
[00131]         for file in lisp-file
[00132]         do
[00133]           (eval (compose-make-latex-command file
[00134]                                             :complete? (when first?
[00135]                                                          complete?)
[00136]                                             :toc? toc?
[00137]                                             :options options
[00138]                                             :options-set? options-set?
[00139]                                             :latex-file latex-file
[00140]                                             :output-directory output-directory
[00141]                                             :title title
[00142]                                             :title-set? title-set?
[00143]                                             :author author
[00144]                                             :author-set? author-set?
[00145]                                             :date date
[00146]                                             :date-set? date-set?))
[00147]           (setq first? nil)))
[00148]    (t ;; then there is just the one output file:
[00149]     (cond
[00150]      ((and output-directory
[00151]            (or (probe-directory output-directory)
[00152]                (error "Output directory ~s is not really a directory." output-directory)))
[00153]       (setq latex-file
[00154]         (concatenate 'string
[00155]                      (string-right-trim "/" output-directory)
```

```
[00156]                       "/"
[00157]                       (if (and latex-file
[00158]                                (not (probe-directory latex-file)))
[00159]                           latex-file
[00160]                         (latex-filename lisp-file)))))
[00161]      ((and latex-file (probe-directory latex-file))
[00162]       (setq latex-file (concatenate 'string
[00163]                          (string-right-trim "/" latex-file)
[00164]                          "/"
[00165]                          (latex-filename lisp-file))))
[00166]      ((null latex-file)
[00167]       (setq latex-file (latex-filename lisp-file))))
[00168]
[00169]
[00170]     (format t "~&Beginning work...~%")
[00171]
[00172]     (format t "~&Reading ~a..." lisp-file)
[00173]     (read-input lisp-file)
[00174]     (format t "done.~%Inserting any defined chunks...")
[00175]     (process-inserted-chunks)
[00176]     (format t "done.~%Writing LaTeX file...")
[00177]     (with-open-file (out latex-file :direction :output
[00178]                      :if-exists :supersede
[00179]                      :if-does-not-exist :create)
[00180]       (when (or complete? (find-embedded-command-data "complete"))
[00181]         (write-header out
[00182]                       toc?
[00183]                       packages
[00184]                       options options-set?
[00185]                       title title-set?
[00186]                       author author-set?
[00187]                       date date-set?))
[00188]       (process-lines out)
[00189]       (when complete?
[00190]         (write-footer out latex-file)))
[00191]     (format t "done.~%Output file is in ~a." latex-file)
[00192]     t
[00193]     )
[00194]    )
[00195]   )
[00196]
[00197]
```

7.3 Inputting and Parsing the Lisp File

Function read-input

The function `read-input` takes one argument, a filename, and it reads that into the `*lines*` variable. While doing this, it looks for embedded commands, and, if it finds one, puts that into the `*embedded-commands*` association list, with the key and value based on the kind of command. (See **??**.) The exception is when the following command is found:

```
<insert name="foo">
```

in which case, that command is placed into `*lines*` so that we know where to put the inserted text.

```
[00198] (defun read-input (file)
```

First reset the global variable `*lineno*`, which tracks the current line:

```
[00199]   (setq *lineno* 0
[00200]         *lines* nil
[00201]         *embedded-commands* nil
[00202]         *index-entries* nil
[00203]         *no-lisp-yet* t
[00204]         *defines-contain-nested-commands* nil
[00205]         )
[00206]   (with-open-file (in file :direction :input)
[00207]     (loop with line
[00208]         while (not (eql :eof (setq line (read-lplisp-input in))))
[00209]         do
```

Convert any ;;[comments to `;;` comments; ditto for ;;] to become `;;`:

```
[00210]           (setq line (replace-re line "^\\s*;+\\[" ";;"))
[00211]           (setq line (replace-re line "^\\s*;+\\]" ";;"))
[00212]           (cond
[00213]             ((not (embedded-command? line))
[00214]              (add-to-lines line))
[00215]             (t
```

Then this is another kind of embedded command. Note that `parse-embedded-command` may read more lines from the stream `in`.

```
[00216]              (parse-embedded-command line in)
[00217]              )))))
[00218]
[00219]
```

Function parse-embedded-command

This takes two arguments, the current line and a stream. If the line contains an embedded command that spans input lines, then the function will read the stream until it finds the end of the current embedded command. In any case, the embedded command will be parsed and put in the `*embedded-commands*` alist. The entry in the alist depends on the kind of embedded command found. So, for example, if it finds the following

```
<define name="foo">
LaTeX commands...
...
</define>
```

it would make this entry into the alist:

```
((define "foo") "LaTeX commands..." ...)
```

Similarly,

```
Toc value="t"
```

would put

`(toc t)`

into the alist.

```
[00220] (defun parse-embedded-command (line stream)
[00221]   (let* ((command (embedded-command line))
[00222]          (function (and command (command-function command))))
[00223]     (cond
[00224]      ((null function)
[00225]       (lplisp-error "Unknown command in: ~s." line))
[00226]      (t
[00227]       (funcall function command stream line)))))
[00228]
[00229]
```

Function process-inserted-chunks

This function reads through `*lines*` and inserts definitions where they are called for. The result is put in `*lines*` again.

```
[00230] (defun process-inserted-chunks ()
[00231]   (let (kind)
[00232]     (setq *lineno* 0)
[00233]     (setq *lines*
[00234]       (loop with target
[00235]           for line in *lines*
[00236]           do (incf *lineno*)
[00237]           when (multiple-value-setq (target kind)
[00238]                  (insert-command? line))
[00239]           append (find-insert-target target kind)
[00240]           else collect line))))
[00241]
```

Function insert-command?

`insert-command?` returns nil if the argument is not an insert command, else it returns the name of the target. Insert commands can be <`insert name="..."`> or <`insert-chunk name="..."`>. This returns the name of the thing to be inserted plus another value that is either `:insert` or `:insert-chunk`, depending on the type of insert command.

```
[00242] (defun insert-command? (line)
[00243]   (let ((command (embedded-command line :anywhere? t)))
[00244]     (when (and command
[00245]                (match-re "^insert" (command-name command)))
[00246]       (values (command-part "name" command)
[00247]               (if (match-re "^insert-chunk" (command-name command))
[00248]                   :insert-chunk
[00249]                 :insert)))))
[00250]
[00251]
[00252]
```

Function find-insert-target

`find-insert-target` will get the target of the insert command (as specified in the argument) and return it as a list of strings.

```
[00253] (defun find-insert-target (target kind)
[00254]   (let* ((tag (if (eql :insert kind)
[00255]                   "define"
[00256]                 "chunk"))
[00257]          (target-val
[00258]           (loop for entry in *embedded-commands*
[00259]              when (and (listp (car entry))
[00260]                        (string-equal (caar entry) tag)
[00261]                        (string-equal (cadar entry) target))
[00262]              return (cdr entry))))
[00263]     (cond
[00264]      ((null target-val)
[00265]       (lplisp-error "No definition for insertion target ~s."
[00266]                     target))
[00267]      ((eql :insert kind)
[00268]       target-val)
[00269]      (t
[00270]       (cons
[00271]        (string-append "<<" (strip-quotes target) ">>=")
```

(string-append ";;
par
noindent
bf
lplispChunk" target "
)

```
[00272]        target-val)))))
[00273]
[00274]
```

Function embedded-command?

The function `embedded-command?` returns non-nil if the line contains an embedded command. It returns the embedded command or nil if there was none. Note that `embedded-command` just calls this function. If `anywhere?` is set, then the embedded command can be anywhere in the line; otherwise, it must be the first non-whitespace, non-comment thing on the line. If `name` is set, then just the command name is returned.

```
[00275] (defun embedded-command? (line &key anywhere? name)
[00276]   (let* ((anywhere-regexp "(<[^>]*>)(.*)")
[00277]          (first-of-line-regexp "^\\s*(#\\||;+)?\\s*(<[^>]*>)(.*)")
[00278]          (match (match-re
[00279]                  (if anywhere?
[00280]                      anywhere-regexp
[00281]                    first-of-line-regexp)
[00282]                  line :return :match))
[00283]          (command (and match (re-submatch match nil nil
[00284]                                           (if anywhere? 1 2)))))
```

If no command-like thing was found, then return nil. However, it's possible that we found something that looks like a command (e.g., <foobar>) that isn't. In that case, command will contain that thing, so we need to skip past it in the line and try again, recursively, if `anywhere?` is set.

```
[00285]     (cond
[00286]      ((null command) nil)
[00287]      ((valid-embedded-command? command)
[00288]       (if name
[00289]           (command-name command)
[00290]         command))
[00291]      (anywhere?
[00292]       (embedded-command? (re-submatch match nil nil
[00293]                                       (if anywhere? 2 3))
[00294]                          :anywhere? anywhere?)))))
[00295]
```

Function embedded-command

```
[00296] (defun embedded-command (line &key anywhere?)
[00297]   (embedded-command? line :anywhere? anywhere?))
[00298]
```

Function valid-embedded-command?

The `valid-embedded-command?` function returns t if its argument is one of the valid embedded command types, as defined in `*valid-embedded-commands*`.

```
[00299] (defun valid-embedded-command? (thing)
[00300]   (let ((command (command-name thing)))
[00301]     (when command
```

Handle case of standalone HTML-like tags, such as toc/:

```
[00302]       (setq command (string-right-trim "/" command))
[00303]       (assoc command *valid-embedded-commands*
[00304]              :test #'(lambda (cmd key)
[00305]                        (cond
[00306]                         ((atom key)
[00307]                          (string-equal cmd key))
[00308]                         (t
[00309]                          (member cmd key
[00310]                                  :test #'string-equal)))))))
[00311]
```

Function command-name

`command-name` returns the actual command name from its argument, assumed to be a command.

```
[00312] (defun command-name (command)
[00313]   (let ((match (and command
[00314]                     (match-re "^\\<([^\\s>]*)" command :return :match))))
[00315]     (and match
[00316]          (re-submatch match nil nil 1))))
[00317]
```

Function command-function

`command-function` takes an embedded command as its argument and returns the function (as specified in `*valid-embedded-commands*`) to handle it, or nil if none exists.

```
[00318] (defun command-function (command)
[00319]   (cdr (assoc (string-right-trim "/"        ; Get rid of possible trailing slash
[00320]                                  (command-name command))
[00321]               *valid-embedded-commands*
[00322]               :test #'(lambda (name key)
[00323]                         (cond
[00324]                          ((atom key)
[00325]                           (string-equal name key))
[00326]                          (t (member name key
[00327]                                     :test #'string-equal)))))))
[00328]
[00329]
[00330]
```

Function command-part

This function, `command-part`, looks for the part specified in the command. For example, in the command

`<define name="foobar">`

the name is `"foobar"`, and this could be returned with:

`(command-part "name" "<define name=f̈oobar¨>")`

```
[00331] (defun command-part (part command)
[00332]   (let ((match (match-re (format nil
[00333]                                  "\\s+~a\\s*=\\s*([^/]*)/?(\\s+|>)"
[00334]                                  part) command :return :match)))
[00335]     (when match
[00336]       (re-submatch match nil nil 1))))
[00337]
```

Functions break?, debug?, debug-off?

These functions return t if a break, debug, or debug-off embedded command occurs in `line`. Note that we break up the commands not due to Lisp's needs, but for when we run LP/Lisp on itself – otherwise, it will process the break, etc., when it reads these functions!

```
[00338] (defun break? (line)
[00339]   (match-re (concatenate 'string "<" "break" ">") line))
[00340]
[00341] (defun debug? (line)
[00342]   (match-re (concatenate 'string "<" "debug" ">") line))
[00343]
[00344] (defun debug-off? (line)
[00345]   (match-re (concatenate 'string "<" "debug-off" ">") line))
[00346]
```

Functions explicit-documentation-start-tag?, explicit-documentation-end-tag?

These functions return t if the command (a string) they are passed are start or end (respectively) tags for explicit comments.

```
[00347] (defun explicit-documentation-start-tag? (cmd)
[00348]   (car (member (string-right-trim ">" (string-left-trim "<" cmd))
[00349]                *documentation-tags*
[00350]                :test #'(lambda (a b)
[00351]                          (string-equal (string-upcase a)
[00352]                                        (string-upcase b))))))
[00353]
[00354] (defun explicit-documentation-end-tag? (cmd)
[00355]   (car (member (string-right-trim ">" (string-left-trim "</" cmd))
[00356]                *documentation-tags*
[00357]                :test #'(lambda (a b)
[00358]                          (string-equal (string-upcase a)
[00359]                                        (string-upcase b))))))
[00360]
```

Functions explicit-documentation-start?, explicit-documentation-end?

Returns non-nil if the string contains the start/end of a long comment. What they actually return is the tag itself.

```
[00361] (defun explicit-documentation-start? (line)
[00362]   (let* ((cmd (embedded-command? line :anywhere? t))
[00363]          (tag (and cmd (command-name cmd))))
[00364]     (and tag
[00365]          (explicit-documentation-start-tag? tag))))
[00366]
[00367] (defun explicit-documentation-end? (line)
[00368]   (let* ((cmd (embedded-command? line :anywhere? t))
[00369]          (tag (and cmd (command-name cmd))))
[00370]     (and tag (explicit-documentation-end-tag? tag))))
[00371]
[00372]
```

Function explicit-documentation-end?

7.4 Writing the Header and Footer of the LaTeX File

The `write-header` function writes the header of the LaTeX file. The keyword parameters provide information about what to put in the header. If they are specified, they take precedence over any corresponding embedded command. The various `xxx-set?` parameters to this function are set to `t` by the caller if its keyword parameters were set by the user. So, for example, if `title` is `nil` but `title-set?` is also `nil`, then we know that the user didn't specify a title, and the embedded title, if any, should be used. However, if `title-set?` is `t`, then the user told the caller explicitly that the title parameter should be `nil`, and so no title should be output at all.

```
[00373] (defun write-header (stream toc? packages
[00374]                      options options-set?
[00375]                      title title-set?
[00376]                      author author-set?
[00377]                      date date-set?)
[00378]   (write-documentclass-line stream options options-set?)
[00379]   (write-preamble stream  packages title title-set?
[00380]                   author author-set?
[00381]                   date date-set? toc?)
[00382]   )
[00383]
```

This writes the `documentclass` command of the LaTeX file. If options are specified, they are included so that the ones specified in the call to `make-latex` are last in the options list, so that they hopefully will have precedence over the ones embedded. `options-set?` is true if the user explicitly set this parameter, which was a keyword parameter in one of the callers.

```
[00384] (defun write-documentclass-line (stream options options-set?)
[00385]   (let ((embedded-options (cdr (find-embedded-command-entry "options"))))
[00386]     (lplisp-output stream "\\documentclass~a{article}~%"
[00387]             (cond
[00388]              ((and (null options) options-set?) "") ;user says no options
[00389]              ((null (or options embedded-options)) ;no options anywhere
[00390]               "")
[00391]              (t
[00392]               (string-append "[" embedded-options options "]"))))
[00393]     )
[00394]   )
[00395]
[00396]
[00397]
```

The `write-preamble` function writes the preamble of the LaTeX file to the output stream. The preamble includes things like the title, author, and date, as well as any other packages or preamble commands specified by the user. Packages, title, author, and date can be specified both in the call to `make-latex` as well as by embedded commands. Other parts of the preamble (e.g., LaTeX macro definitions) can be specified only by embedded commands. Where there is a conflict, resolution is in favor of the arguments supplied in the call to `make-latex`. (Packages are additive, however, with the ones from embedded commands coming first.) In addition, any strings in `*standard-preamble*` are included first. After writing the begin-document command, a `maketitle` command is inserted, if there was a title specified. If a table of contents was requested, then the title is placed on a title page first.

```
[00398] (defun write-preamble (stream packages title title-set?
```

```
[00399]                       author author-set?
[00400]                       date date-set? toc?)
[00401]   (let (data title?)
[00402]     (loop for line in *standard-preamble*
[00403]         do
[00404]           (lplisp-output stream line)
[00405]           )
[00406]
[00407]     (when (setq data (find-embedded-command-data "packages"))
[00408]       (lplisp-output stream
[00409]                      (string-append "\\usepackage{"
[00410]                                     data
[00411]                                     "}~%")))
[00412]
[00413]     (when packages
[00414]       (lplisp-output stream
[00415]                      (string-append "\\usepackage{"
[00416]                                     packages
[00417]                                     "}~%")))
[00418]
[00419]     (when (setq data (find-embedded-command-data "preamble"))
[00420]       (lplisp-output stream "~%%% begin preamble from embedded commands:~%")
[00421]       (loop for line in data
[00422]           do
[00423]             (lplisp-output stream line))
[00424]       (lplisp-output stream "%% end preamble from embedded commands~%~%"))
[00425]
[00426]
[00427]     (unless (and title-set? (null title))
[00428]       (when (setq data (or title
[00429]                            (find-embedded-command-data "title")))
[00430]         (setq title? t)
[00431]         (when (listp data)
[00432]           (setq data (apply #'concatenate (cons 'string data))))
[00433]         (lplisp-output stream "\\title{~a}~%" data)))
[00434]
[00435]     (unless (and author-set? (null author))
[00436]       (when (setq data (or author (find-embedded-command-data "author")))
[00437]         (when (listp data)
[00438]           (setq data (apply #'concatenate (cons 'string data))))
[00439]         (lplisp-output stream "\\author{~a}~%" data)))
[00440]
[00441]     (unless (and date-set? (null date))
[00442]       (when (setq data (or date (find-embedded-command-data "date")))
[00443]         (when (listp data)
[00444]           (setq data (apply #'concatenate (cons 'string data))))
[00445]         (lplisp-output stream "\\date{~a}~%" data)))
[00446]
[00447]     (lplisp-output stream "~%\\makeindex~%")
[00448]     (lplisp-output stream "~%\\begin{document}~%~%")
[00449]
[00450]     (when title?
```

```
[00451]       (cond
[00452]        ((null toc?)
[00453]         (lplisp-output stream "\\maketitle\\thispagestyle{empty}~%"))
[00454]        (t
[00455]         (lplisp-output stream "\\begin{titlepage}~%")
[00456]         (lplisp-output stream "\\maketitle\\thispagestyle{empty}~%")
[00457]         (lplisp-output stream "\\end{titlepage}~%~%")
[00458]         (lplisp-output stream "\\tableofcontents~%\\newpage~%"))))
[00459]     )
[00460]   )
[00461]
```

`write-footer` is pretty simple – it just ends the LaTeX file, after writing the indexing stuff. Since the indexing depends on a `.ind` file, this also touches that file in the current directory.

```
[00462] (defun write-footer (stream latex-filename)
[00463]   (let ((base (file-basename latex-filename)))
[00464]     (lplisp-output stream
[00465]                    (concatenate 'string
[00466]                      "%% Index:~%"
[00467]                      "%% To create an index, run makeindex on ~a.idx to produce~%"
[00468]                      "%% ~a.ind.  Then uncomment the following lines.~%"
[00469]                      "%%~%"
```

”

input a.ind

```
[00470]                      "%\\cleardoublepage~%"
[00471]                      "%\\addcontentsline{toc}{section}{Index}~%"
[00472]                      "%\\printindex~%"
[00473]                      "\\end{document}~%")
[00474]           base base base)
```

We have to make sure that base.ind exists:

```
[00475]   (with-open-file (out (concatenate 'string base ".ind")
[00476]                    :direction :probe
[00477]                    :if-does-not-exist :create))))
[00478]
[00479]
```

7.5 Converting from LP/Lisp to LaTeX

Function process-lines

The function `process-lines` is where most of the work of formatting the LP/Lisp file occurs. After some initial set up, the first thing that this function does is to find out what mode we're in. At any particular time, the current mode can be in a long comment, in a short comment, processing (non-comment) Lisp code, and so forth. All non-blank, non-comment lines are considered Lisp code that should be typeset in a pretty manner for the reader, prefaced by line numbers. Unless we are in Lisp or `longComment` modes, we skip blank lines. This prevents us from spuriously entering and leaving Lisp mode when there are blank lines between comments. Within lines of Lisp code, there can be comments as well. These can be tricky for LP/Lisp to find, since although they are set off

by semicolons, there can be semicolons within strings or specified as characters in Lisp itself. This means that the regular expression to find these has to take that into account. Consequently, we ignore these for now. A future feature might find these and typeset them nicely.

```
[00480] (defun process-lines (stream)
[00481]   (setq *codeLineNo* 1                        ;reset code counter
[00482]         *lineno* 0                          ;and line counter
[00483]         *processing-explicit-documentation* nil)
[00484]   (loop with line = (next-line)
[00485]       while line
[00486]       do
[00487]         (cond
[00488]          ((or (and *implicit-documentation*
[00489]                    (long-comment-start? line))
[00490]               (long-documentation-start? line))
[00491]           (setq line (process-long-comment line stream)))
[00492]          ((or (and *implicit-documentation*
[00493]                    (short-comment? line))
[00494]               (short-documentation-start? line))
[00495]           (setq line (process-short-comment line stream)))
[00496]          ((blank-line? line)
```

If we encounter a blank line that isn't in a long comment or within non-comment Lisp code, just skip it.

```
[00497]           (setq line (next-line)))
[00498]          ((long-comment-end? line)
```

This shouldn't happen, since this is supposed to be handled within `process-long-comment` or `process-lisp-code`.

```
[00499]           (lplisp-error "Encountered unexpected end of long comment."))
[00500]          (t
```

Otherwise, assume we're now in Lisp mode. The `process-lisp-code` function has to read beyond the end of the Lisp code in order to determine that we're not in Lisp mode any longer, so `line` will be the next line after the Lisp code when it returns:

```
[00501]           (setq line (process-lisp-code line stream))
[00502]           )
[00503]          )))
[00504]
```

Function process-long-comment

The `process-long-comment` function handles the case when we enter a long comment, which is signaled by the #| characters. This will read the lines from the next line until it encounters the comment end characters (|#), processing each one. If `*implicit-documentation*` is set, then each line is processed as LaTeX source; all explicit documentation tags are ignored. If not, then each line is processed as Lisp code until/unless an explicit documentation tag (e.g., <doc>) is encountered. At that point, we leave Lisp mode (if we were in it) and process lines as LaTeX until either the end of the long comment is found or the corresponding close tag is found. This is done by the helper function `process-long-comment-explicit-docs`. On entry, `line` is the comment start, and on exit, it will be either the rest of the line after the end characters or the next line.

```
[00505] (defun process-long-comment (line stream)
[00506]   (cond
[00507]    ((not *implicit-documentation*)
[00508]     (process-long-comment-explicit-docs line stream))
[00509]    (t
[00510]     (let ((match (match-re "#\\|\\s*(.+)" line :return :match)))
[00511]
```

If there is text on the first line, then use it, else read the next line:

```
[00512]       (setq line
[00513]         (cond
[00514]          (match
[00515]           (re-submatch match nil nil 1)
[00516]           )
[00517]          (t
[00518]           (next-line))))
[00519]       (loop while (and line (not (long-comment-end? line)))
[00520]           do
[00521]             (process-latex-line line stream)
[00522]             (setq line (next-line)))
```

If we exit the loop without finding the end of the long comment, it's an error. Otherwise, we need to get the next line or the remaining text on the current line.

```
[00523]       (cond
[00524]        ((null line)
[00525]         (lplisp-error "End of file encountered when inside a long comment."))
[00526]        ((setq match (match-re "\\|#\\s*(.+)" line :return :match))
[00527]         (re-submatch match nil nil 1))
[00528]        (t
[00529]         (next-line)))))))
[00530]
```

Process-long-comment-explicit-docs

```
[00531] (defun process-long-comment-explicit-docs (line stream)
[00532]   (let ((doc-start (explicit-documentation-start-tag?
[00533]                     (embedded-command line :anywhere? t))))
[00534]     (cond
[00535]      (*processing-explicit-documentation*
[00536]       (lplisp-error
[00537]        (concatenate 'string
[00538]          "Illegal long Lisp comment inside of explicit documentation block!"
[00539]          "~%Try putting the documentation start tag inside the long comment.")))
[00540]      (t
```

This is the start of an explicit documentation chunk. We shouldn't be in Lisp mode now, since we require the start tag to be on the same line as the long comment start characters, and `process-lisp-code` should have kicked us out of Lisp code mode when it saw the line. Now, change mode to currently processing explicit documentation, and process the rest of the comment as LaTeX.

```
[00541]       (setq *processing-explicit-documentation* t
[00542]             line (next-line))
[00543]       (process-lce-docs-aux line stream :doc-start doc-start)))))
[00544]
[00545]
[00546] (defun process-lce-docs-aux (line stream &key doc-start)
[00547]   (cond
[00548]    ((null line)                                ;end of file -- shouldn't happen!
[00549]     (lplisp-error
[00550]      "Something's wrong: encountered EOF while processing long comment.")
[00551]     nil)
[00552]    ((long-comment-end? line)
[00553]     (setq *processing-explicit-documentation* nil)
[00554]     (next-line)                                 ;just return the next line
[00555]     )
[00556]    (t                                         ;process the line as \LaTeX{}
[00557]     (process-latex-line
[00558]      (replace-re line
[00559]                  (concatenate 'string
[00560]                    "</" doc-start ">")
[00561]                  "")
[00562]      stream)
[00563]     (process-lce-docs-aux (next-line) stream :doc-start doc-start))))
[00564]
[00565]
```

Function process-short-comment

If `*implicit-documentation*` is set, or if this is not the start of an explicit comment, this just strips off any leading semicolons and white space, deletes any embedded commands, and processes the rest of the line as LaTeX source. It does nothing to `*lines*`. If this is the start of an explicit documentation line or set, then we have output this line as LaTeX, but also continue processing lines until we encounter the end tag for the documentation or we encounter something that is not a comment. At that point, we return that line.

```
[00566] (defun process-short-comment (line stream)
[00567]   (let ((doc-start (explicit-documentation-start? line)))
```

First output the current line, if there's anything on it:

```
[00568]     (setq line (replace-re
[00569]                 (replace-re line "^\\s*;+\\s*" "")
[00570]                 (concatenate 'string
[00571]                   "</?" doc-start ">")
[00572]                 ""))
[00573]     (unless (string-equal "" line)
[00574]       (process-latex-line line stream))
[00575]
[00576]     (setq line (next-line))
[00577]
```

Now, if `*implicit-documentation*` is nil, continue until we run out of comments or hit the end tag; else return.

```
[00578]     (cond                                ;shouldn't really hit first case...
[00579]      (*implicit-documentation*
[00580]       line)
[00581]      (t
[00582]       (loop with rest
[00583]           until (or (null line)
[00584]                     (not (short-comment? line)))
[00585]           when (explicit-documentation-end? line)
[00586]             return
[00587]              (let (semis)
```

We've encountered the end of the explicit documentation; we have to now output anything that may have come before the tag, then retun everything afterward as a comment to the caller.

```
[00588]                (setq rest (split-re (string-append "</" doc-start
[00589]                                                    ">") line)
[00590]                    semis (match-re "^(;*)[^;]" line :return :match))
[00591]              (process-latex-line (eat-semicolons (car rest)) stream)
[00592]              (setq line (string-append (re-submatch semis nil nil 1)
[00593]                                        " " (cadr rest)))
[00594]              )
[00595]             else do
[00596]              (process-latex-line
[00597]               (replace-re
[00598]                (replace-re line "^\\s*;+\\s*" "")
[00599]                (concatenate 'string
[00600]                  "</?" doc-start ">")
[00601]                "")
[00602]               stream)
[00603]              (setq line (next-line)))
[00604]       line))))
[00605]
[00606]
[00607]
[00608]
```

Function process-latex-line

This function does the bulk of the work in processing a line of LaTeX source from a comment form to that suitable for processing by `latex`. About all this does at the moment is to replace [[[[foo]]]] with
`textttfoo`, and to call `translate-tildes`. The latter is needed not due to LaTeX, but rather because of `lplisp-output` – if the LaTeX line has a tilde in it, it'll confuse `format`, which `lplisp-output` calls. It then writes the line to `stream`.

```
[00609] (defun process-latex-line (line stream)
[00610]   (setq line (translate-tildes (translate-function-headings
[00611]                                 (translate-markup line))))
[00612]   (lplisp-output stream (if (string-equal "" line)
[00613]                             "~%"
[00614]                           line)))
[00615]
[00616]
```

Function process-lisp-code

The function `process-lisp-code` takes a line of Lisp code as its first argument. It first opens a `lispMode` LaTeX environment, then translates some special characters and outputs the line, prefaced by its line number (in the non-comment Lisp code, not the line in the source file). It does this for all other non-comment Lisp lines. When it encounters a short or long comment, what it does depends on whether or not the user specified explicit or implicit documentation. If implicit, then this exits lisp mode (currently `verbatim`) and returns immediately; the comment line will be returned. If explicit, then if the comment is the start of an explicit documentation long comment or if it is a short explicit documentation, then it exits Lisp mode and returns the line; if it is any other kind of comment, it processes it as Lisp code and continues. If `*implicit-documentation*` is nil, then this will also ignore any commented lines starting on column 1 – the intent is to ignore comment header blocks – which should be in LaTeX in any case – and also commented-out blocks of code. In-line comments are preserved and output.

```
[00617] (defun process-lisp-code (line stream)
[00618]   (cond
[00619]    ((or (blank-line? line)
[00620]         (column-1-comment? line))
[00621]     (next-line))
[00622]    (t
[00623]     (let (pending-index old-line)
[00624]       (start-lisp-mode stream)
[00625]       (loop
[00626]           while (continue-processing-lisp-code? line)
[00627]           do
[00628]             (cond
[00629]              ((column-1-comment? line)
[00630]               (setq old-line line)
[00631]               (setq line (next-line)))
[00632]              ((and *no-lisp-yet*
[00633]                    (blank-line? line))
[00634]               (setq old-line line)
[00635]               (setq line (next-line)))
[00636]              (t
[00637]               (lplisp-output stream
[00638]                              "[~5,'0d] ~a~%"
[00639]                              *codeLineNo*
[00640]                              (process-lisp-line line))
[00641]               (setq old-line line)
[00642]               (setq line (next-line))
[00643]               (incf *codeLineNo*)
[00644]               (setq *no-lisp-yet* nil)
[00645]               )))
[00646]       (end-lisp-mode stream)
[00647]       line)
[00648]     )
[00649]    )
[00650]   )
[00651]
[00652]
[00653]
[00654] (defun continue-processing-lisp-code? (line)
```

```
[00655]   (cond
[00656]    ((null line) nil)
[00657]    ((and (not (long-comment-start? line))
[00658]          (not (short-comment? line)))
[00659]     t)
[00660]    ((or *implicit-documentation*
[00661]         (explicit-documentation-start? line))
[00662]     nil)
[00663]    (t t)))
[00664]
```

Function start-lisp-mode

This function inserts LaTeX code to start the Lisp mode. The corresponding end function is `end-lisp-mode`.

```
[00665] (defun start-lisp-mode (stream)
[00666]   (lplisp-output stream
[00667]                  "~&{\\small~%\\begin{alltt}~%"))
[00668]
[00669]
[00670]
[00671]
```

Function end-lisp-mode

The actual line to end verbatim mode is not shown here because it is commented out for LP/Lisp – if this was not the case, LaTeX would think that this line itself is ending verbatim when we run LP/Lisp on this file!

```
[00672] (defun end-lisp-mode (stream)
[00673]   (lplisp-output stream  ;; omitted line follows here in source file
[00674]                  )
[00675]   )
[00676]
[00677]
[00678]
[00679]
```

Function process-lisp-line

`process-lisp-line` processes the line of Lisp code by translating any tabs to spaces. (This could be done using the `moreverb` LaTeX package, but doing it here keeps us from having to load another package.) We are currently using the `verbatim` environment to typeset Lisp code. If we were to switch to something more flexible, such as `alltt`, we would need to do additional translation of LaTeX special characters. This is trickier than it seems—an early version of LP/Lisp attempted this, with mixed success. Some translation code for that is still in the source file, commented out, if needed in the future. This does the following:

- Translate double square bracketed expressions that may appear in comments. In this case, we translate them into `textit`, since they will already be within text typeset in `texttt`.

- Escape dollar signs and ampersands.
- Escape, using math mode, > and <. Note that we need to do this *after* escaping dollar signs, or the ones we insert for math mode will become escaped.
- Reader macros for characters need to be escaped as well.

We also keep track, for indexing purposes, of where things are defined. We'll also boldface the things being defined.

```
[00680] (defun process-lisp-line (line)
[00681]   (add-index (translate-lisp-line line)))
[00682]
[00683]
[00684]
```

Function translate-lisp-line

The translation functions, starting with `translate-lisp-line`, user `replace-re` to replace some characters with valid LaTeX source. However, `translate-tildes` is needed not due to LaTeX, but rather because of `lplisp-output` – if the LaTeX line has a tilde in it, it'll confuse `format`, which `lplisp-output` calls.

```
[00685] (defun translate-lisp-line (line)
[00686]   (setq line (translate-tabs line))
[00687]   (cond
[00688]    (*allow-embedded-latex*
[00689]     line)
[00690]    (t (escape-latex-characters line))))
[00691]
[00692]
```

Function escape-latex-characters(line)

This function will "escape" LaTeX characters in `line`, i.e., protect them from being interpreted as LaTeX. **Function escape-latex-characters(line)**

This will escape most LaTeX characters in `line`. Note that the functions called by this, `escape-XXX`, are not complete—they may insert the symbol `LPBRACEPAIR` where they want {} to show up; consequently, `replace-bracepairs` is called afterward to clean up.

```
[00693] (defun escape-latex-characters (line)
[00694]   (replace-bracepairs
[00695]    (escape-braces
```

It's important that backslashes are escaped first—otherwise, the backslashes that may be inserted by escaping braces (e.g.) would be escaped, thus undoing what the other function did!

```
[00696]     (escape-ampersands
[00697]      (escape-backslashes line)))))
[00698]
[00699] (defun escape-ampersands (line)
[00700]   (replace-re line "&" "\\&"))
[00701]
```

```
[00702] (defun escape-backslashes (line)
[00703]   (replace-re line "\\\\" "\\textbackslash"))
[00704]
[00705] (defun replace-bracepairs (line)
[00706]   (replace-re line "" "{}"))
[00707]
[00708] (defun escape-braces (line)
[00709]   (setq line (replace-re line "\\{" "\\textbraceleft"))
[00710]   (replace-re line "\\}" "\\textbraceright"))
[00711]
[00712]
[00713]
```

Function translate-tabs

This function will expand tabs into spaces, with `*tab-width*` spaces for each tab.

```
[00714] (defun translate-tabs (line)
[00715]   (replace-re line "\\t"
[00716]               (make-string *tab-width* :initial-element #\space)))
[00717]
[00718]
[00719]
```

Function replace-slashes-in-strings

`replace-slashes-in-strings` replaces any backslash in a string with double backslashes. This is done so that we can have LaTeX commands in the comments at the end of the lines, but we can still use backslashes in (e.g.) regular expressions.

```
[00720] (defun replace-slashes-in-strings (line)
[00721]   (replace-re line
[00722]                "\"[^\"]+\""
```

This regexp finds a string. Now we need to look inside of it, using another `replace-re` call, to find slashes to replace. Note that we do not replace slashes immediately followed by a curly brace, since those have likely been inserted by `translate-braces`.

```
[00723]                #'(lambda (match)
[00724]                    (replace-re (car (last match))
[00725]                                "\\\\."
[00726]                                #'(lambda (m)
[00727]                                   (setq m (car (last m)))

[00728]                                   (cond
[00729]                                    ((member (char m (- (length m) 1))
[00730]                                             '(#\} #\{))

[00731]                                     m)
[00732]                                    ((string-equal (string #\\)
[00733]                                                   (char m
[00734]                                                         (- (length m) 1)))
```

```
[00735]                                      "\\textbackslash\\textbackslash{}")
[00736]                                    (t

[00737]                                     (concatenate 'string "\\textbackslash{}"
[00738]                                                  (string
[00739]                                                   (char m
[00740]                                                         (- (length m) 1)))))
[00741]                                    ))))))
[00742]
```

Function translate-tildes

```
[00743] (defun translate-tildes (line)
[00744]   (replace-re line "~" "~~"))
[00745]
[00746]
```

Function add-index

The `add-index` function will check to see if the current Lisp line defines anything, which is assumed if the line starts with `(def`. If so, an index entry is added for the definition.

```
[00747] (defun add-index (line)
[00748]   (let* (
[00749]          (match (match-re "^\\((def\\S*)\\s+([^\\s(]*)\\s*(.*)"
[00750]                           line :return :match))
[00751]          (def (and match (re-submatch match nil nil 1)))
[00752]          (name (and match (re-submatch match nil nil 2)))
[00753]          (index-entry (and match
[00754]                            (format nil "\\index{~a (~a; code line ~d)}"
[00755]                                    name (definition-type def) *codeLineNo*)))
[00756]          pos)
[00757]     (cond
[00758]      ((not match) line)
[00759]      (t
[00760]        (setq index-entry (replace-re index-entry
[00761]                                      "_"
[00762]                                      "\\lplispHyphen{}"))
[00763]        (setq index-entry (replace-re index-entry
[00764]                                      ">"
[00765]                                      "\\lplispRArrow{}"))
[00766]       (concatenate 'string
[00767]         (subseq line 0 (setq pos (search name line)))
[00768]         "\\underline{"
[00769]         name
[00770]         "}"
[00771]         (subseq line (+ (length name) pos) )
[00772]         index-entry)))))
[00773]
[00774]
[00775]
[00776]
[00777]
```

Function translate-function-headings

This replaces <`function...`>, etc., with the appropriate section heading. Note that due to some weirdness in the way <`define...`> is handled, any function heading within a define needs to not span multiple lines!

```
[00778] (defun translate-function-headings (line)
[00779]   (replace-re line "<([^>]+)>"
[00780]              #'(lambda (match-pair)
[00781]                  (format nil
[00782]                          "~%\\bigskip\\leftline{\\textbf{~a}}~%\\medskip"
[00783]                          (string-upcase (cadr match-pair) :end 1)))))
[00784]
[00785]
[00786]
```

Function translate-markup

The `translate-markup` function takes a string and replaces all double square bracket expressions with the appropriate LaTeX markup, as discussed in Section 2.

```
[00787] (defun translate-markup (line)
[00788]   (replace-re line "\\[\\[([^\\]]*)\\]\\]"
[00789]       #'(lambda (match-pair)
[00790]           (let ((section? (match-re "^(\\*+)\\s+(.+)\\s\\**$"
[00791]                                     (car (last match-pair))
[00792]                                     :return :match)))
[00793]             (cond
[00794]              ((not section?)
[00795]               (string-append "\\texttt{" (car (last match-pair))
[00796]                              "}"))
[00797]              (t
[00798]               (string-append
[00799]                "\\"
[00800]                (cond
[00801]                 ((string-equal "*" (re-submatch section? nil nil 1))
[00802]                  "section")
[00803]                 ((string-equal "**" (re-submatch section? nil nil 1))
[00804]                  "subsection")
[00805]                 ((string-equal "***" (re-submatch section? nil nil 1))
[00806]                  "subsubsection")
[00807]                 ((string-equal "****" (re-submatch section? nil nil 1))
[00808]                  "paragraph")
[00809]                 ((string-equal "*****" (re-submatch section? nil nil 1))
[00810]                  "subparagraph")
[00811]                 (t
[00812]                  (lplisp-error "Too many levels of subsection requested.")))
[00813]                "{"
[00814]                (re-submatch section? nil nil 2)
[00815]                "}")))))))
[00816]
```

7.6 Embedded Command Parsers

All embedded commands except `insert` are parsed by their own functions, as specified in `*valid-embedded-commar`
Each is passed three arguments: the command (a string), the current input stream (since some commands can span lines), and the line the command was on. **Function embedded-insert**

The `embedded-insert` command is called when an `insert` command is encountered. Unlike most of the other commands, it does not modify `*embedded-commands*`, but instead makes an entry into `*lines*` of the command itself, so that later we can replace it with the actual thing requested. The only tricky part is what kind of line this comment is part of. If it is on the first line of a long comment, then we need to put the long comment characters in `*lines*` first, then the command. If it is part of a short comment, then we need to strip off the leading semicolons.

```
[00817] (defun embedded-insert (command stream line)
[00818]   (embedded-handle-long-comment line)
[00819]   (add-to-lines command))
[00820]
```

Function embedded-define

The `embedded-define` command finds the name of the thing being defined, then searches for the end tag (`</define>`) in order to build the definition (as a list of lines), then adds this to the `*embedded-commands*` alist. If `#|<define ...>` occurs on a line by itself, then don't output the long comment start characters; similarly for `</define>|#`.

```
[00821] (defun embedded-define (command stream line)
[00822]   (let ((name (command-part "name" command))
[00823]         entry)
[00824]     (cond
[00825]      ((null name)
[00826]       (lplisp-error "Define does not have a name!"))
[00827]      (t
[00828]       (unless (match-re "#\\|<define" line)
[00829]         (embedded-handle-long-comment line))
[00830]       (collect-multiline-embedded-command "define" stream line :name name
[00831]                                           :allow-nested? t)
[00832]       ))))
[00833]
[00834]
```

Function embedded-chunk

This is very much like `<define>`, except that it puts a marker for the chunk directly in the Lisp code at this point so that the reader understands where the missing chunk should go. The idea is to provide the same sort of mechanism literate programming tools like `noweb` do to talk about code in pieces that may be out of order. The corresponding `<insert-chunk>` command is used to typeset the chunk later.

```
[00835] (defun embedded-chunk (command stream line)
[00836]   (let ((name (command-part "name" command))
[00837]         entry)
[00838]     (cond
[00839]      ((null name)
```

```
[00840]       (lplisp-error "Chunk does not have a name!"))
[00841]      (t
[00842]       (unless (match-re "#\\|<chunk" line)
[00843]         (embedded-handle-long-comment line))
[00844]       (collect-multiline-embedded-command "chunk" stream line :name name
[00845]                                           :allow-nested? t)
```

Now, we need to mark that a chunk has been taken out:

```
[00846]       (add-to-lines (format-chunk-marker name line))
[00847]       ))))
[00848]
```

Function format-chunk-marker

This returns <<`chunk-name`>> to insert into the Lisp code at this point. What is actually returned also has the same amount of leading spaces (or whitespace in general) as the line this came from.

```
[00849] (defun format-chunk-marker (chunk-name line)
[00850]   (let ((match (match-re "(^\\s*)\\S" line :return :match)))
[00851]     (string-append (re-submatch match nil nil 1)
[00852]                    "<<" (strip-quotes chunk-name) ">>")))
[00853]
[00854]
[00855]
```

Function embedded-insert-chunk

This just calls `embedded-insert` to put the command at the end of `*lines*` to be processed during the next phase.

```
[00856] (defun embedded-insert-chunk (command stream line)
[00857]   (embedded-insert command stream line))
[00858]
[00859]
[00860]
```

Function embedded-toc

For `toc` and `complete` embedded commands, we just add entries to the `*embedded-commands*` alist saying that they were there (i.e., the cdr is `t`).

```
[00861] (defun embedded-toc (command stream line)
[00862]   (embedded-handle-long-comment line)
[00863]   (add-to-embedded-commands "toc" t))
[00864]
```

Function embedded-complete

```
[00865] (defun embedded-complete (command stream line)
[00866]   (embedded-handle-long-comment line)
[00867]   (add-to-embedded-commands "complete" t))
[00868]
```

For most other embedded commands, we just collect the possibly multiline text and put into the alist. **Function embedded-title**

```
[00869] (defun embedded-title (command stream line)
[00870]   (embedded-handle-long-comment line)
[00871]   (collect-multiline-embedded-command "title" stream line))
[00872]
```

Function embedded-author

```
[00873] (defun embedded-author (command stream line)
[00874]   (embedded-handle-long-comment line)
[00875]   (collect-multiline-embedded-command "author" stream line))
[00876]
```

Function embedded-date

```
[00877] (defun embedded-date (command stream line)
[00878]   (embedded-handle-long-comment line)
[00879]   (collect-multiline-embedded-command "date" stream line))
[00880]
```

Function embedded-packages

```
[00881] (defun embedded-packages (command stream line)
[00882]   (embedded-handle-long-comment line)
[00883]   (collect-multiline-embedded-command "packages" stream line
[00884]                                       :eat-semicolons? t))
[00885]
```

Function embedded-options

```
[00886] (defun embedded-options (command stream line)
[00887]   (embedded-handle-long-comment line)
[00888]   (collect-multiline-embedded-command "options" stream line
[00889]                                       :eat-semicolons? t))
[00890]
```

Function embedded-preamble

```
[00891] (defun embedded-preamble (command stream line)
[00892]   (embedded-handle-long-comment line)
[00893]   (collect-multiline-embedded-command "preamble" stream line
[00894]                                       :eat-semicolons? t))
[00895]
```

Function embedded-function

`embedded-function` handles commands such as <`function` *name*> It handles functions, methods, etc.

```
[00896] (defun embedded-function (command stream line)
[00897]   (embedded-handle-long-comment line)
[00898]   (let ((match (match-re "<([^\\s]*)\\s+(.*)>" command :return :match)))
[00899]     (loop for thing in (list ";;;\\bigskip"
[00900]                              (concatenate 'string
[00901]                                ";;; \\leftline{\\textbf{"
[00902]                                (string-capitalize
[00903]                                 (re-submatch match nil nil 1))
[00904]                                " "
[00905]                                (re-submatch match nil nil 2)
[00906]                                "}}"
[00907]                                (string #\newline)
[00908]                                "\\medskip"
[00909]                                )
[00910]                              ";;;"
[00911]                              ";;;"
[00912]                              )
[00913]         do (add-to-lines thing))))
[00914]
[00915]
```

Function embedded-ignore

The function `embedded-ignore` does the same as the others, but the entry in `*embedded-commands*` is never used.

```
[00916] (defun embedded-ignore (command stream line)
[00917]   (embedded-handle-long-comment line)
[00918]   (collect-multiline-embedded-command "ignore" stream line))
[00919]
```

Function embedded-comment

For comments, we want to read until the end of the comment, but we don't need to save the text.

```
[00920] (defun embedded-comment (command stream line)
[00921]   (embedded-handle-long-comment line)
[00922]   (cond
[00923]    ((end-tag-on-line? "comment" line))
[00924]    (t
[00925]     (loop until (or (eql (setq line (read-lplisp-input stream)) :EOF)
[00926]                     (end-tag-on-line? "comment" line))))))
[00927]
```

Function embedded-unexpected-tag

The `embedded-unexpected-tag` is called when an embedded command is encountered, but it's not what we expected. This might happen, for example, if we have one too many closing tags for a command.

```
[00928] (defun embedded-unexpected-tag (command stream line)
[00929]   (lplisp-error "Unexpected command ~s -- possibly closing one too many tags?"
[00930]                 command))
[00931]
```

Function embedded-implicit-documentation-function

Function embedded-implicit-documentation-function

These functions set the value of `*implicit-documentation*`, thus controlling whether the user has to explicitly mark comments for inclusion in LaTeX or not.

```
[00932] (defun embedded-implicit-documentation-function (command stream line)
[00933]   (ignore command stream line)
[00934]   (setq *implicit-documentation* t))
[00935]
[00936] (defun embedded-explicit-documentation-function (command stream line)
[00937]   (ignore command stream line)
[00938]   (setq *implicit-documentation* nil))
[00939]
```

Function embedded-nop-function

This just puts the line as-is in to `*lines*`.

```
[00940] (defun embedded-nop-function (command stream line)
[00941]   (ignore command stream)
[00942]   (add-to-lines line))
[00943]
```

Function remove-embedded-commands

This removes embedded commands from a line.

```
[00944] (defun remove-embedded-commands (line)
[00945]   (replace-re line "<[^>]*>" ""))
[00946]
[00947]
[00948]
```

Should only encounter if `*implicit-documentation*`, in which case just remove the command and add the line to `*lines*`.

```
[00949] (defun embedded-doc-closing-tag (command stream line)
[00950]   (let (match)
[00951]     (cond
[00952]      ((null *implicit-commands*)
[00953]       (embedded-unexpected-tag command stream line))
[00954]      (t
[00955]       (add-to-lines (remove-embedded-commands line))))))
[00956]
[00957]
```

Function embedded-handle-long-comment

This checks to see if `line` is the beginning of a long comment (i.e., starts with #|). If so, then it "outputs" the start-of-comment characters one `*lines*`.

```
[00958] (defun embedded-handle-long-comment (line)
[00959]   (when (long-comment-start? line)
[00960]     (setq *lines* (append-to-end "#|" *lines*))))
[00961]
```

Function embedded-break-read, embedded-break

Function embedded-debug, embedded-debug-off

These functions allow the user to turn LP/Lisp's debugging facilities on/off. They are probably only useful for debugging LP/Lisp itself. A <`break-read`> command is handled by `embedded-break-read`, which calls Lisp's `break` right there. <`break`> causes a break to occur during `next-line`. <`debug`> and <`debug-off`> turn debugging on and off.

```
[00962] (defun embedded-break-read (command stream line)
[00963]   (ignore command stream)
[00964]   (break "Breaking at user request during read.")
[00965]   )
[00966]
[00967] (defun embedded-break (command stream line)
[00968]   (ignore command stream)
[00969]   (add-to-lines line))
[00970]
[00971] (defun embedded-debug (command stream line)
[00972]   (ignore command stream)
[00973]   (add-to-lines line))
[00974]
[00975] (defun embedded-debug-off (command stream line)
[00976]   (ignore command stream)
[00977]   (add-to-lines line))
[00978]
[00979]
```

Function append-to-end

7.7 Utility and Helper Functions

This function, `append-to-end`, will put `thing` on the end of `list` destructively. This differs from `cons-end`, which is defined in the local utilities, in that the return value is what matters. It returns the modified list if it wasn't null to begin with. If it was null, then it returns a new list containing the single element `thing`. So consequently, you should always plan on catching and using the return value.

```
[00980] (defun append-to-end (thing list)
[00981]   (cond
[00982]    ((not (null list))
[00983]     (setf (cdr (last list)) (list thing))
[00984]     list)
[00985]    (t
[00986]     (list thing))))
[00987]
```

Function definition-type

This returns the kind of thing (as a string) defined by the argument. For example, if "defun" is passed in, "function" is passed back.

```
[00988] (defun definition-type (def)
[00989]   (case (intern (string-upcase def))
[00990]     (defun "function")
[00991]     (defclass "class")
[00992]     (defvar "variable")
[00993]     (defparameter "parameter")
[00994]     (defmethod "method")
[00995]     (defgeneric "generic function")
[00996]     (defmacro "macro")
[00997]     (otherwise def)))
[00998]
[00999]
[01000]
```

Function strip-leading-semicolons

The function `strip-leading-semicolons` does just that, also removing any leading whitespace from the line. This is how we turn short Lisp comments into LaTeX code. This uses the regexp2 functions `match-re` and `re-submatch`; see those functions' documentation for details.

```
[01001] (defun strip-leading-semicolons (line)
[01002]   (let ((match (match-re "^\\s*;+\\s*(.*)$" line :return :match)))
[01003]     (cond
[01004]      ((null match)
[01005]       line)
[01006]      (t
[01007]       (re-submatch match nil nil 1)))))
[01008]
```

Function read-lplisp-input

`read-lplisp-input` reads a line from a stream (passed as its argument), returning either the line or `:EOF` if we have reached the end of the stream. It also updates the global variable `*lineno*` to track which line we have just read.

```
[01009] (defun read-lplisp-input (stream)
[01010]   (let ((line (read-line stream nil :eof)))
[01011]     (unless (eql :eof line)
[01012]       (incf *lineno*))
[01013]     line))
[01014]
```

Function next-line

`next-line` gets the next line from `*lines*` and increments the line counter, `*lineno*`. This also handles debugging, including handling embedded debugging commands.

```
[01015] (defun next-line ()
[01016]   (let ((next (pop *lines*)) doc)
[01017]     (cond
[01018]      ((null next)
[01019]       (when *debug-lplisp*
[01020]         (format t "~&[LP/LISP: Trying to read past end of *lines*]~%"))
[01021]       nil)
[01022]      ((break? next)
[01023]       (break "Breaking at user request, line ~5,'0d." *lineNo*)
[01024]       (next-line))
[01025]      ((debug? next)
[01026]       (format t "~&(turning debugging on)~%")
[01027]       (debug-lplisp t)
[01028]       (next-line))
[01029]      ((debug-off? next)
[01030]       (format t "~&(turning debugging off)~%")
[01031]       (debug-lplisp nil)
[01032]       (next-line))
[01033]      (t
[01034]       (incf *lineno*)
[01035]       (when (zerop (mod *lineNo* 10))
[01036]         (princ "."))
[01037]       (when *debug-lplisp*
[01038]         (format t "~&[LP/LISP: Ready to process the following line:~%~5t~a~%"
[01039]                 next)
[01040]         (format t "~&[LP/LISP: Press return to continue...")
[01041]         (read-line)
[01042]         (fresh-line))
```

If `*implicit-documentation*` is set, then get rid of any embedded documentation commands:

```
[01043]       (cond
[01044]        ((not *implicit-documentation*)
[01045]         next)
[01046]        ((not (setq doc (or (explicit-documentation-start? next)
[01047]                            (explicit-documentation-end? next))))
[01048]         next)
[01049]        (t
[01050]         (replace-re next (concatenate 'string
[01051]                            "</?" doc ">")
[01052]                     ""))))))))
[01053]
```

/?" doc "

") **Function latex-filename**

The `latex-filename` function takes a filename and returns a LaTeX filename based on it.

```
[01054] (defun latex-filename (file)
```

The local variable `match` is a match object created by `match-re`. The regular expression groups everything but whatever follows the final period in the filename. `re-submatch` just returns that submatch. Note that if there was no match, `match` is nil, and so that case has to be handled.

```
[01055]   (let ((base (file-basename file)))
[01056]     (cond
```

It's an error if `file` is null.

```
[01057]      ((null file)
[01058]       (error "Can't find LaTeX filename based without a Lisp filename." ))
```

If there is no filename extension on the original Lisp file, then use that as the base for the LaTeX filename.

```
[01059]      ((null base)
[01060]       (concatenate 'string file ".tex"))
```

Otherwise, use the base filename + `.tex`.

```
[01061]      (t
[01062]       (concatenate 'string base ".tex")))))
[01063]
```

Function file-basename

Return the base name of a filename – that is, the file name minus the extension, if any.

```
[01064] (defun file-basename (filename)
[01065]   (let ((match (match-re "(.*)\\.[^.]*$" filename :return :match)))
[01066]     (cond
[01067]      (match
[01068]       (re-submatch match nil nil 1))
[01069]      (t filename))))
[01070]
[01071]
[01072]
```

Function long-comment-start?

`long-comment-start?`, will return t if the line starts with the long comment start characters; `long-comment-end?` returns t if the line closes a long comment.

```
[01073] (defun long-comment-start? (line)
[01074]   (match-re "^#\\|" line))
[01075]
```

Function long-comment-end?

```
[01076] (defun long-comment-end? (line)
[01077]   (match-re "\\|#" line))
[01078]
[01079]
```

Function short-comment?

This will return non-nil if the argument, a string, starts with semicolons (after possibly some whitespace). The thing it actually returns is a `regexp2` match object. In this case, submatch 1 will be the text (minus leading whitespace) of the comment.

```
[01080] (defun short-comment? (line)
[01081]   (match-re "^\\s*;+\\s*(.*)" line :return :match))
[01082]
```

Function blank-line?

This returns true if the string it is passed is empty or full of white space.

```
[01083] (defun blank-line? (line)
[01084]   (match-re "^\\s*$" line))
[01085]
[01086]
[01087]
```

Function collect-multiline-embedded-command

Many embedded command parsers process commands that can span multiple lines. `collect-multiline-embedded-`

will look from the current line forward in the stream until the corresponding end tag is found, then add an entry to the `*embedded-commands*` alist whose key is the command name (as a string) and whose value is a list of the lines between the tags. This also handles the case of text on the line after the start command, before the end command, or both. Unless `allow-nested?` is true, it also signals an error if a start or end command is found that is not expected – i.e., no nested embedded commands are allowed. If `name` is specified, then the actual key used in `*embedded-commands*` is a list,

`(tag name)`

If `eat-semicolons?` is set, then we get rid of leading semicolons on the lines. This is useful, for example, for `package` or `option` (or anything else) that are in short comments rather than long ones, but that span multiple lines.

```
[01088] (defun collect-multiline-embedded-command (tag stream line &key name prefix
[01089]                                                                 eat-semicolons?
[01090]                                                                 allow-nested?)
[01091]   (let (definition matched? text match)
[01092]     (multiple-value-setq (matched? text)
[01093]       (end-tag-on-line? tag line :after-start-tag? t))
[01094]     (cond
```

If `matched?` is true, then the termination tag is on this line, making our life possibly easier; however, we need to make sure there are no nested commands inside of `text`

```
[01095]      (matched?
[01096]       (if (or allow-nested?
[01097]               (not (embedded-command? text :anywhere? t)))
[01098]         (setq definition (list text))
[01099]         (lplisp-error
[01100]          "Found an embedded command within another one, which isn't allowed.")))
```

Otherwise, check to see if there is another embedded command on this line – if so, that's an error:

```
[01101]      ((and (not allow-nested?)
[01102]            (setq match (match-re ">(.+)" line :return :match))
[01103]            (embedded-command?
[01104]             (re-submatch match nil nil 1) :anywhere? t))
[01105]       (lplisp-error
[01106]        "Found an embedded command within another one, which isn't allowed."))
[01107]      (t
```

Otherwise, we need to find the termination tag by reading the stream. First, though, we check to see if there is additional text on this line; if so, we need ;; to add that to the definitions:

```
[01108]       (when (setq match (match-re ">\\s*(.+)\\s*" line :return :match))
[01109]         (setq definition (append-to-end
[01110]                           (eat-semicolons (re-submatch match nil nil 1)
[01111]                                           eat-semicolons?)
[01112]                           definition)))
[01113]       (loop with done? = nil
[01114]           while (not (eql (setq line (read-lplisp-input stream)) :EOF))
[01115]           do
[01116]             (cond
[01117]              ((not (multiple-value-setq (matched? text)
[01118]                      (end-tag-on-line? tag line)))
```

No end tag found, so just collect the line, as long as there is no embedded command in it:

```
[01119]               (if (or allow-nested?
[01120]                       (not (embedded-command? line :anywhere? t)))
[01121]                   (setq definition (append-to-end
[01122]                                     (eat-semicolons line eat-semicolons?)
[01123]                                     definition))
[01124]                 (lplisp-error
[01125]                  "Found an embedded command within another one, which isn't allowed.")
[01126]                 ))
[01127]              (t
```

Found the end tag. If there was text returned, then we put that on the end of the definition list. Then exit the loop. If the text contains an embedded command, however, we have to signal an error:

```
[01128]               (when (and text (not (string-equal text "")))
[01129]                 (if (not (embedded-command? text :anywhere? t))
[01130]                     (setq definition (append-to-end
[01131]                                       (eat-semicolons text eat-semicolons?)
```

```
[01132]                                        definition))
[01133]                   (lplisp-error
[01134]                    (string-append
[01135]                     "Found an embedded command within another one, "
[01136]                     "which isn't allowed.")))
[01137]                 )
[01138]               (setq done? t)))
[01139]           until done?)))
```

Now update the `*embedded-commands*` alist with an entry for the tag:

```
[01140]     (add-to-embedded-commands
[01141]      (if name
[01142]          (list tag name)
[01143]        tag)
[01144]      definition)))
[01145]
[01146]
[01147]
[01148]
```

Function end-tag-on-line?

A common problem for the embedded command parsers is to see if an end tag is on the current line.

This function, `end-tag-on-line?`, checks for this. Its first argument is the tag (e.g., `"define"`), and the second argument is the line. It returns two values. The first is whether or not the end tag was found, and the second is any text that occurs before the end tag. If the keyword argument `after-start-tag?` is set, then this explicitly looks for the end tag after another tag. Note that it returns nil if there is not a start tag followed by some (optional) text followed by an end tag.

```
[01149] (defun end-tag-on-line? (tag line &key after-start-tag?)
[01150]   (let ((match (match-re
[01151]                 (format nil
[01152]                         "~a(.*)\\s*</~a>"
[01153]                         (if after-start-tag?
[01154]                             ">\\s*"
[01155]                           "")
[01156]                         tag)
[01157]                 line :return :match)))
[01158]     (values (not (null match))
[01159]             (and match (re-submatch match nil nil 1)))))
[01160]
[01161]
[01162]
```

Function lplisp-error

`lplisp-error` is the error function that should be called when LP/Lisp finds an error in an input line. It outputs lines prefaced with the line number on which the error was found.

```
[01163] (defun lplisp-error (format-string &rest args)
[01164]   (apply #'error (cons (format nil "Line ~s: ~a"
[01165]                                *lineno* format-string)
[01166]                        args)))
[01167]
[01168]
```

Function add-to-embedded-commands

This function will add a new entry (or replace the existing entry) for "tag" to the `*embedded-commands*` alist. The value (as the `cdr`), will be the second argument.

```
[01169] (defun add-to-embedded-commands (tag definition)
[01170]   (let ((entry (find-embedded-command-entry tag)))
[01171]     (cond
[01172]      ((null entry)
[01173]       (push (cons tag definition) *embedded-commands*))
[01174]      (t
[01175]       (setf (cdr entry) definition)))))
[01176]
```

Function add-to-lines

This adds the argument to the `*lines*` global variable.

```
[01177] (defun add-to-lines (line)
[01178]   (cond
[01179]    ((null *lines*)
[01180]     (setq *lines* (list line)))
[01181]    (t
[01182]     (setf (cdr (last *lines*)) (list line)))))
[01183]
[01184]
[01185]
```

Function find-embedded-command-entry

This function returns the entry in the `*embedded-commands*` alist corresponding to its first argument. This should be a string. Each alist key is either a string or a list whose first element is a string (e.g., `(("define" "foobar") l1 l2...)`). The command itself can be either a string or a list. For example,

`("define" "ïntroduction)̈`

is a perfectly fine command name for the definition labeled "introduction".

```
[01186] (defun find-embedded-command-entry (command)
[01187]   (assoc command *embedded-commands*
[01188]          :test #'equal))
[01189]
```

Function find-embedded-command-data

`find-embedded-command-data` returns the data portion of the embedded command specified.

```
[01190] (defun find-embedded-command-data (command)
[01191]   (cdr (find-embedded-command-entry command)))
[01192]
```

Function debug-lplisp

The `debug-lplisp` function, when called with no arguments, turns single-stepping and debugging on. When called with nil, it turns it off.

```
[01193] (defun debug-lplisp (&optional (on? t))
[01194]   (setq *debug-lplisp* on?))
[01195]
```

Function lplisp-output

This function is the output function for LP/Lisp. It takes the same arguments as `format`, which it calls, and sends the formatted output either to `stream` or `*standard-output*`, depending on whether debugging is off or on, respectively.

```
[01196] (defun lplisp-output (stream string &rest args)
[01197]   (let ((out (apply #'format (cons nil (cons string args)))))
[01198]     (princ out stream)
[01199]     (fresh-line stream)
[01200]     (when *debug-lplisp*
[01201]       (princ out)
[01202]       (fresh-line))))
[01203]
[01204]
[01205]
```

Function compose-make-latex-command

This function creates a command to call `make-latex` that can be eval'd to run LP/Lisp. It is used for creating recursive calls in the `make-latex` function. It is used to create a call with just those keyword parameters specified that had values set in the caller, as shown by the `title-set?`, etc., keywords.

```
[01206] (defun compose-make-latex-command (file &key
[01207]                                            complete?
[01208]                                            toc?
[01209]                                            options
[01210]                                            options-set?
[01211]                                            latex-file
[01212]                                            output-directory
[01213]                                            title
[01214]                                            title-set?
[01215]                                            author
[01216]                                            author-set?
[01217]                                            date
[01218]                                            date-set?
[01219]                                            (documentation :implicit))
[01220]   (cons 'make-latex
[01221]         (cons file
[01222]               (append (list :documentation documentation)
[01223]                       (remove :ignore
[01224]                       `(:complete? ,complete?
[01225]                         :toc? ,toc?
```

```
[01226]                          ,@(if (not options-set?)
[01227]                                '(:ignore)
[01228]                               (list :options options))
[01229]                          :latex-file ,(if output-directory
```

Then ignore the `latex-file` variable, and create a new filename based on `output-directory` and the Lisp file name. If not, then just use the current directory.

```
[01230]                                          (concatenate 'string
[01231]                                            (string-right-trim "/" output-directory)
[01232]                                            "/"
[01233]                                            (latex-filename file))
[01234]                                         (if (probe-directory latex-file)
```

If the `latex-file` is a directory, then use that directory:

```
[01235]                                            (concatenate 'string
[01236]                                              (string-right-trim "/" latex-directory)
[01237]                                              "/"
[01238]                                              (latex-filename file))
[01239]                                           nil))
[01240]                          ,@(if (not title-set?)
[01241]                                '(:ignore)
[01242]                               (list :title title))
[01243]                          ,@(if (not author-set?)
[01244]                                '(:ignore)
[01245]                               (list :author author))
[01246]                          ,@(if (not date-set?)
[01247]                                '(:ignore)
[01248]                               (list :date date))))))))
[01249]
```

Function lisp-line?

This returns t if the line passed to it is neither the start of a long comment or a short comment.

```
[01250] (defun lisp-line? (line)
[01251]   (and line
[01252]        (not (long-comment-start? line))
[01253]        (not (short-comment? line))))
[01254]
```

Function long-documentation-start?, short-documentation-start?

These will return non-nil if `line` is the start of a long comment or a short comment (respectively) that also has a start documentation tag on it.

```
[01255] (defun long-documentation-start? (line)
[01256]   (and (long-comment-start? line)
[01257]        (explicit-documentation-start? line)))
[01258]
[01259]
[01260] (defun short-documentation-start? (line)
[01261]   (and (short-comment? line)
[01262]        (explicit-documentation-start? line)))
[01263]
[01264]
```

Function column-1-comment?

This returns t if there is a semicolon in column 1 of the line.

```
[01265] (defun column-1-comment? (line)
[01266]   (match-re "^;.*" line))
[01267]
```

Function write-shell-script

This function will write the `lplisp` shell script (or whatever you want to name it) that runs LP/Lisp from the command line. This is done so that the correct image name can be inserted in the header line. To use this, give it the name you want the file to have (as the `name` keyword parameter), an image name, if different from the one currently running, and the directory in which the lplisp.fasl file will live (which defaults to the current directory).

```
[01268] (defun write-shell-script (&key (name "lplisp")
[01269]                                 image-name
[01270]                                 (LPLisp-directory (path-namestring (pwd))))
[01271]   (with-open-file (script (concatenate 'string
[01272]                             (string-right-trim "/" LPLisp-directory)
[01273]                             "/"
[01274]                             name)
[01275]                   :direction :output :if-exists :supersede
[01276]                   :if-does-not-exist :create)
[01277]     (format script
[01278]             "#! ~a -#!~%~%" (or image-name (car (sys:command-line-arguments))))
[01279]     (format script "
[01280] (with-open-file (*standard-output* \"/dev/null\" :direction :output
[01281]                   :if-exists :append)
[01282]   (without-redefinition-warnings
[01283]    (load \"~a/lplisp.fasl\")))
[01284] (format t \"~~&This is LP/Lisp -- Literate Programming in Lisp.~~%\")
[01285]
[01286]
[01287]
[01288]
[01289] (defun implode-with-commas (string-list)
[01290]   (apply #'concatenate
[01291]                     (cons 'string
[01292]                           (cons (car string-list)
[01293]                                 (mapcar #'(lambda (s)
[01294]                                             (concatenate 'string
[01295]                                               \",\"
[01296]                                               s))
[01297]                                         (cdr string-list))))))
[01298]
[01299] (defun doopt (name slist llist)
[01300]   (if (or slist llist)
[01301]       (format nil \":~~s ~~s\" name
[01302]               (implode-with-commas (append slist llist)))
[01303]     \"\"))
[01304]
```

```
[01305]
[01306] (sys:with-command-line-arguments
[01307]     ((\"toc\" :long toc)
[01308]      (\"c\" :short complete)
[01309]      (\"complete\" :long lcomplete)
[01310]      (\"o\" :short options :allow-multiple-options :required-companion)
[01311]      (\"option\" :long loptions :allow-multiple-options :required-companion)
[01312]      (\"p\" :short packages :allow-multiple-options :required-companion)
[01313]      (\"package\" :long lpackages :allow-multiple-options :required-companion)
[01314]      (\"l\" :short latex-file :required-companion)
[01315]      (\"latex\" :long llatex-file :required-companion)
[01316]      (\"t\" :short title :required-companion)
[01317]      (\"title\" :long ltitle :required-companion)
[01318]      (\"a\" :short author :required-companion)
[01319]      (\"author\" :long lauthor :required-companion)
[01320]      (\"d\" :short date :required-companion)
[01321]      (\"date\" :long ldate :required-companion)
[01322]      (\"e\" :short e-docs)
[01323]      (\"explicit\" :long le-docs)
[01324]      (\"i\" :short i-docs)
[01325]      (\"implicit\" :long li-docs)
[01326]      )
[01327]   (rest :command-line-arguments (sys:command-line-arguments)
[01328]         :long-abbrev t)
[01329]
[01330]   (when (and (or e-docs le-docs)
[01331]              (or i-docs li-docs))
[01332]     (error \"Can't specify both explicit and implicit documentation!\"))
[01333]
[01334] (let ((command
[01335]         (read-from-string (format nil \"
[01336] (lplisp:make-latex ~~a
[01337]        ~~a     ;;toc
[01338]        ~~a     ;;complete
[01339]        ~~a     ;;options
[01340]        ~~a     ;;packages
[01341]        ~~a     ;;latex-file
[01342]        ~~a     ;;title
[01343]        ~~a     ;;author
[01344]        ~~a     ;;date
[01345]        ~~a     ;;documentation type
[01346]      )
[01347] \"
[01348]
```

if there is more than one Lisp file specified, then call with that as a list; otherwise just use the file specified:

```
[01349]           (if (cddr rest)
[01350]             (format nil \"'~~s\" (cdr rest))
[01351]           (format nil \"~~s\" (second rest)))
[01352]           (if toc \":toc? t\" \"\")
[01353]           (if (or complete lcomplete)
```

```
[01354]               \":complete? t\"
[01355]             \"\")
[01356]           (doopt 'options options loptions)
[01357]           (doopt 'packages packages lpackages)
[01358]           (if (or latex-file llatex-file)
[01359]               (format nil \":latex-file ~~s\"
[01360]                       (or latex-file llatex-file))
[01361]             \"\")
[01362]           (if (or title ltitle)
[01363]               (format nil \":title ~~s\" (or title ltitle))
[01364]             \"\")
[01365]           (if (or author lauthor)
[01366]               (format nil \":author ~~s\" (or author lauthor))
[01367]             \"\")
[01368]           (if (or date ldate)
[01369]               (format nil \":date ~~s\" (or date ldate))
[01370]             \"\")
[01371]           (if (or e-docs le-docs)
[01372]               \":documentation :explicit\"
[01373]               (if (or i-docs li-docs)
[01374]                 \":documentation :implicit\"
[01375]                \"\"))))))
[01376]
[01377] (eval command)
[01378]
[01379] ))
[01380]
[01381] (format t \"~~%\")
[01382]
[01383]
[01384] "
[01385]     (string-right-trim "/" LPLisp-directory)
[01386]             )
[01387] )
[01388] )
[01389]
```

Function string-append

Append all arguments into one string. If some of the things are lists, then the strings they contain are appended.

```
[01390] (defun string-append (&rest things)
[01391]   (labels ((sa (strings)
[01392]            (cond
[01393]             ((null strings)
[01394]              "")
[01395]             ((stringp (car strings))
[01396]              (concatenate 'string (car strings)
[01397]                           (sa (cdr strings))))
[01398]             (t
[01399]              (concatenate 'string (sa (car strings)) (sa (cdr strings)))))))
[01400]     (sa things)))
```

```
[01401]
```

Function eat-semicolons

This function takes two arguments, a string and a Boolean. If the Boolean second argument is true, then this eats all semicolons from the front of the first argument, otherwise it just returns.

```
[01402] (defun eat-semicolons (string &optional (really? t))
[01403]   (if (not really?)
[01404]       string
[01405]     (string-left-trim ";" string)))
[01406]
[01407]
```

Function blank-line

This returns t if the `line` it is passed is empty.

```
[01408] (defun blank-line? (line)
[01409]   (match-re "^\\s*$" line))
[01410]
```

Function strip-quotes

This takes the quotes off the both sides of `thing`.

```
[01411] (defun strip-quotes (thing)
[01412]   (string-trim "\"" thing))
[01413]
[01414]
[01415]
```