LP/Lisp: Literate Programming for Lisp

Roy M. Turner
Department of Computer Science
University of Maine
Orono, ME 04469–5752
rmt@umcs.maine.edu

ABSTRACT

Writing a program and writing its documentation are often considered two separate tasks, leading to several problems: the documentation may never be written; when it is, it may be an afterthought; and when the program is modified, the needed changes to the documentation may be overlooked. *Literate programming* (LP), introduced by Donald Knuth, views a program and its documentation as an integrated whole: they are written together to inform both the computer and human readers. LP tools then extract the code for the computer and the documentation for further document processing. Unfortunately, existing LP tools are much more suited for compiled languages, where there is already a step between coding and executing and debugging the code. Lisp programming typically involves incremental development and testing, often highly interleaving coding with running portions of the code. Thus LP tools inject an artificial impediment into this process.

LP/Lisp is a new LP tool designed specifically for Lisp and the usual style of programming using Lisp. The literate programming file *is* the Lisp file; LP markup and text resides in Lisp comments, where it does not interfere with running the code. LP/Lisp provides the usual literate programming services, such as code typesetting, syntactic sugaring, and the ability to split the code for expository purposes (a "chunk" mechanism). LP/Lisp, itself written in Lisp, is run on the code to produce the documentation.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—Documentation

General Terms

Documentation, Languages

Keywords

Lisp, Literate Programming, LaTeX

Writing a program and writing its documentation are often considered two separate tasks. This leads to at least three problems. First, the documentation is often never written. Second, when it is written, it is often an afterthought. And third, when the program changes, often the documentation is *not* changed, leading to out-of-date and confusing documentation.

Literate programming was introduced by Donald Knuth to address these problems [4]. Literate programming views programs and documentation as a unified whole, and writing both as essentially writing a work of literature. The program and the documentation are written together, with the goal of informing not only the computer, but also the reader. Literate programming (LP) tools then process the combined document to produce either the program, the documentation, or both.

The first LP tool was Knuth's own `WEB` system. The languages in the combined file—which we will call the LP file—were TeX to express the human-oriented material and PASCAL to express the program itself. TeX and PASCAL were freely mixed in the LP file, with some additional notational markers to denote context and to label pieces of the program (which are often referred to as *chunks*).

The LP file itself was neither particularly human-friendly nor compiler-readable. Instead, two programs operate on the LP file. One, `weave`, produces the human-targeted output, in this case, a TeX file that can be formatted and printed. The other, `tangle`, produces a PASCAL program that can be compiled and run. The author does not edit these files him- or herself, but rather edits the LP file. This has the beneficial effect of encouraging documentation and code to change together.

If this were all `WEB` did, it would be useful, but only somewhat more so than documentation tools such as Javadoc [9]. But `WEB` did more. For example, it allows the program to be written in pieces that are interspersed with text describing those pieces. In the human-readable output (produced by `weave`), the code pieces, called "chunks", are typeset as code and can be referred to by name elsewhere, including in other chunks. Chunks can be defined out of order, with respect to how they appear in the source code, for expository purposes. In the program output (produced by `tangle`), the pieces are gathered together and put into the appropriate location and order.

There have been many literate programming tools and languages developed since the original work on WEB. For example, CWEB [5] (for C, C++, and Java), FWEB [6] (for FORTRAN and other scientific programming languages), and noweb [8, 3] (for general-purpose use). Few have been developed specifically for Lisp.

In his paper introducing WEB, Knuth listed Lisp as one of the languages that could be used as the programming portion of the combined document. While this is undoubtedly true, we disagree with the spirit of Knuth's remarks: although Lisp *could* be the programming language portion of an LP tool like WEB, this does not mean that it *should* be. Or, more to the point, LP tools like WEB are not really suitable for interpreted languages like Lisp.[1]

Why isn't, say, an excellent LP tool like noweb adequate for Lisp programs? Let's first look at the style of programming that WEB, noweb, and virtually all other LP tools assume. The author first writes the combined LP file. Then the author runs an LP tool on the file to produce the documentation, and another tool (or the same tool with different switches) to produce the actual program in the target language. This assumes that the programmer is using the write–compile–execute model of programming, and thus it is very well-suited for programs written in PASCAL, C, Java, or any other compiled language.

However, this is not how Lisp programmers operate. We generally write pieces of the program and test them out in the interpreter. This may lead us to change some other pieces, which we do in the IDE (Emacs, for example), then we test those changes out, without ever (or only seldom) leaving the interpreter. This is one thing that gives Lisp its power as a rapid prototyping language.

We do not, then, want to have to make our modifications to an LP file, which is not interpretable, then to have to run the result through something like tangle. This puts an extra step in our prototyping and debugging, a very unnatural, undesirable, compiler-like step.

What we have done with LP/Lisp is to develop a literate programming mechanism that works with the Lisp style of programming. Instead of having a separate LP file that gets processed to produce the documentation and the code, we take the approach of allowing the Lisp file itself to be the LP file. Lisp comments, with appropriate syntactic markers, contain the documentation, written in LaTeX. Mark-up conventions are provided to help hide some of the LaTeX syntax so as not to disturb the look of the Lisp comments—so that for the most part, the comments can serve as documentation for the programmer as he or she is debugging as well as fancier (or at least, more well-formatted) printed documentation. LP/Lisp then reads the LP file to produce the pure LaTeX source file that is then processed in the usual way to produce printed or on-line (e.g., Web-based) documentation. LP/Lisp is itself written in Lisp.

[1]And here, we mean languages really *like* Lisp: ones that are ideal for rapid idea development and prototyping, or even "noodling" around to get a feel for a problem. This rules out languages such as Visual Basic, Java, and, to some extent, Perl.

In the remainder of this paper, we first briefly touch on related approaches to literate programming for Lisp. Then we describe the LP/Lisp model of literate programming for Lisp, including an overview of the approach, a description of the markup understood by the program, and a little bit about the program itself. Finally, we discuss the next version of LP/Lisp, including syntactic changes and changes to the implementation.

1. RELATED WORK

The one WEB-like LP tool specifically for Lisp of which we are aware is CLWEB [7]. CLWEB attempts to overcome the shortcomings of traditional LP approaches for Lisp development. It is a Lisp program that performs tangling and weaving on a literate programming file containing mixed, marked-up, TeX and Lisp code. The syntax of the literate programming file is the same, basically, as cweb.

If this were all CLWEB provided, it would not offer much to recommend it over using cweb itself. However, it also provides a Emacs Lisp file (a ".el" file) that defines a CLWEB mode for use when editing the LP file. Using this mode allows Emacs to understand the LP file enough to be an IDE for CLWEB, much as Emacs is usually used as an IDE for Lisp code. This allows the user to deal with the LP file directly as the source file during development. For example, the mode allows the user to evaluate the code within a section, so that changes can be made and tested without a complete tangle needing to be done.

As elegant as this approach is, it still has some significant drawbacks. A program developed in CLWEB is tied to it for future development of the code. If CLWEB is not available, then the user will have to edit the tangled Lisp file directly, since there is no easy way to use the LP file without CLWEB. This might be a problem, for example, when code is distributed to others for use. It also tightly ties the literate programming process to a particular IDE, Emacs. While Emacs is a very good IDE, it may not be the one preferred by all users. And if it is not, the temptation will almost certainly be to debug, and hence change, the Lisp program by using the tangled Lisp source, not the LP program. In this case, either the Lisp and LP sources will drift apart, or the Lisp source runs the risk of being overwritten by a new tangle.

In addition, while it is likely that much of their functionality would still be available, it is also unclear how CLWEB would integrate easily with other parts of even an Emacs-based IDE, such as SLIME or Allegro Common Lisp's Emacs–Lisp interface.[2] Consider a user encountering an error at run time. Most interfaces to Emacs allow the user to request, from the debugger, that the code corresponding to a function be edited in an Emacs buffer. Even if the function in question was developed in an CLWEB LP file, the interface will almost certainly edit the tangled file instead.

A different approach is taken by Scribble [1] for literate programming in the Scheme language (among other things). Scribble, written in Scheme, basically implements a new

[2]Emacs Lisp code is provided as part of the mode for interaction with SLIME; however, not all of the possible functions of the interface are necessarily supported.

language for dealing with both documentation and Scheme code. It is highly extensible and, at least when paired with an IDE such as DrScheme,[3] provides a natural way to interact with both the textual and code parts of an LP program. It is a very impressive system, so much so, that had we been aware of it when we began work on LP/Lisp, and had a Common Lisp version been readily available, we likely would have used it instead of working on our own.

However, that being said, even this very nice system still suffers from the drawback of needing special-purpose code (Scribble itself) in order to run the Lisp code. While this may not be too much of a problem currently—Scribble is easily accessed and loaded by Scheme—it is potentially a problem should development or distribution of Scribble cease at some point. And, of course, as far as we are aware, Scribble exists only for Scheme, not Common Lisp.

Our approach is similar in some respects to pbook [2], which is an Emacs Lisp program for generating documentation for Emacs Lisp programs.[4] Like LP/Lisp, pbook embeds the documentation in comments, with LP markup. From this, it generates a PDF version of the result.[5]

There are some differences between LP/Lisp and pbook. First, pbook is written in Emacs Lisp, not Common Lisp itself, and so it requires Emacs to work. Second, there is significant additional functionality in LP/Lisp usually found in LP tools (e.g., support for chunks) that is missing in pbook.

2. OVERVIEW OF LP/LISP

We had several requirements for LP/Lisp. First, it should be as unobtrusive as possible for the Lisp programming process. This means that the presence of the LP markup should not detract from the code itself, since the programmer will be most concerned with reading and changing that during the development and debugging processes.

Second, we want the LP file to stand alone as a Lisp file for the reasons discussed above, without the need for special LP-related language (Lisp) extensions. There should be no tangle process; the LP file *is* the Lisp file. A consequence of this is the LP markup should not detract from the ability of the unprocessed comments to help the programmer during programming and debugging. We do not want the programmer to have to refer constantly to printed documentation during programming, but rather to have access to useful comments.

Third, we wanted the ability to use the full range of a text markup (specification) language as well as have full control over the resulting weaved file. The user should not have to learn special markup just for LP, unless he or she desires, but instead should be able to use the text markup language itself. This means, obviously, that the documentation needs to be hidden from Lisp itself, since we do not wish to have to depend on Lisp extensions at run- or compile-time to safely ignore the documentation. This means that the documentation needs to live in the comments or in strings interspersed with the code. We chose comments as the most natural means of incorporating documentation.

For the markup language supported, we chose LaTeX. LaTeX is a very rich, powerful, and widely-used language for specifying printed material. Not only that, but with the use of readily-available tools (e.g., latex2html[6]), LaTeX can serve as an excellent base language from which other formats, such as HTML, can be derived.

Fourth, we wanted to provide functionality similar to the *chunk* facility of `WEB`-like tools such as `noweb`. In these LP languages, it is possible to break code apart for discussion. For example, a function's skeleton might be presented one place in the documentation, with its internal details hidden, but labeled. In a different place, were it makes sense to discuss the details, the detailed code would appear. This is done in most LP tools in a straightforward way, with the order of code definition following that of the documentation. The `tangle` process then puts the chunks in the right order to create the source file.

Since the LP and source (Lisp) files are the same for us, we take a different approach. LP markup defines, in the Lisp source, the start and end of a chunk. These chunks are then elided automatically when the documentation is processed and are included in the documentation only when explicitly referred to.

Finally, we wanted to provide automatically a range of other facilities that would support the creation of documentation from Lisp code. This includes line numbering, typesetting code and comments in different typefaces, automatic indexing of functions, and so forth.

3. LP MARKUP

The LP markup in the current, initial, version of LP/Lisp was influenced both by conventions used in `WEB` and its descendants as well as in markup languages such as XML. XML-like tags were chosen, instead of, for example, s-expressions, for widespread familiarity and because they visually set off the LP markup in the comments. In addition, this type of markup can provide both beginning and ending delimiters, thus making it clear where the documentation (e.g.) begins and ends.[7]

Since all LP markup exists inside of Lisp comments, we first need to discuss comments in general. We distinguish three types of comments: long, full-line, and partial-line. *Long comments* are those begun by the `#|` character sequence and ended by `|#`. Lisp ignores everything between these sets of characters, making this kind of comment very useful, in LP, for long sections of LaTeX code. *Full-line comments* are those lines whose first non-whitespace character is the Lisp comment character (`;`). Several of these lines can occur together, forming blocks of comments or LaTeX text. *Partial-line comments* are those that occur on a line with some Lisp

[3]Now DrRacket; see `racket-lang.org`.
[4]Indeed, our next version of LP/Lisp is beginning to look in some respects like the markup in pbook; see the section on future work, below.
[5][Actually, it generates a LaTeX version; PDF was stated erroneously in the original published paper.]
[6]See `www.latex2html.org`.
[7]Below, we discuss how we are changing this in the next version.

```
;;;
;;; This is a comment.
;;;<doc>
;;; This is not a comment.
;;; Neither is this.
;;; </doc>
;;; But this is.
;;;

(defun foo (bar)
   ;; Comment...
   ;;[Not a comment
   ;;]
   ;; Comment again
   code)
```

This is not a comment. Neither is this.

```
[00001] (defun foo (bar)
[00002]    ;; Comment...
```

Not a comment

```
[00003]    ;; Comment again
[00004]    code)
```

Figure 1: Example of explicit-mode markup of documentation. Left, the LP file. Right, the formatted output.

code.

3.1 Documentation Modes

LP/Lisp has two modes. In *implicit mode*, all long and full-line comments are considered documentation. The leading semicolons, if any, are stripped off and the text, apart from any embedded LP markup, is output directly to the LaTeX file upon processing. In *explicit mode*, special markup tags are used to specify which commented text is to be considered documentation. Full-line comments that are not embedded in Lisp forms are ignored. They can thus be used for information useful during programming that might not be appropriate to include in the documentation or for commenting out sections of Lisp code. All other comments are considered part of the code and are typeset in the same fashion as the code itself.

Figure 1 shows this. Note that the
`<doc>...</doc>`
construction is used to denote documentation. Also note that there is a shorthand form of this:
`;;[...<newline>;;]`
This is provided as a convenience. In actuality, either form is terminated by a non-comment line. Thus, short sections of LaTeX can be included in the documentation as:

```
(defun foo (bar)
   ;;[This will appear in the documentation as
   ;;\LaTeX{}
   (print 'hi)
   ;; And this will appear as a comment in the
   ;; code
 )
```

In either of these examples, LaTeX index entries are created for the function `foo`.

The user can specify implicit or explicit mode either by options to the LP program itself or in the Lisp file itself. In the latter case, this would be done using the `<implicit/>` or `<explicit/>` markup in a comment.[8]

[8]This can be done anywhere in the file, since the entire LP file is read before any output is produced.

The user can, even in implicit mode, cause the LP program to ignore sections of code and/or comments. This is done via the `<comment>...</comment>` or `<ignore>...</ignore>` markup tags. They are identical; alternative forms are provided for the user's convenience. Possible uses for this include skipping details that would be tedious in the documentation, protecting proprietary code from being printed in the documentation, or skipping over comments (in implicit mode) and/or code that only make sense to the Lisp programmer or that is temporary.

3.2 LaTeX-Related Markups

LP markup is provided as well to control LaTeX, that is, to include LaTeX code in the preamble (i.e., before the
`\begin{document}`
line) of the LaTeX file. Some specific markup is provided for common options or commands. For example,
`<title>...</title>`
`<author>...</author>`
`<date>...</date>`
are provided to specify the title, author, and date, respectively. The single tag `<toc/>` is provided to specify a table of contents. There is also a tag, `<complete/>`, provided to tell LP/Lisp to create a complete LaTeX file, including the preamble. If this is not specified (and not requested in the call to the LP program), then only the portion of the document that would appear between the
`\begin{document}...\end{document}`
tags is generated.

LP markup is provided as well to set LaTeX options, such as `12pt`, etc., that appear in the `\documentclass` command. These can be set individually, for example:
`<option>12pt</option>`
`<option>landscape</option>`
or all at once:
`<options>12pt,landscape</options>`

Similarly, markup is provided for including LaTeX, packages:
`<package>alltt</package>`
`<packages>local,html</packages>`
In addition, there is markup (`<preamble>...</preamble>`)

for including arbitrary LaTeX code in the preamble, for example, to define LaTeX macros.

3.3 Syntactic Shortcuts

There are some syntactic sugaring markups, as well. We used the same syntax as (e.g.) `noweb` to allow the programmer to specify that something should be typeset in typewriter text, which is a fairly standard convention used to indicate, e.g., a token of the language being described. For example,

```
This is a [[token]].
```

will produce:

This is a `token`.

on output.We have also provided sectioning markups (cf. `WEB`):

```
[[* This is a section *]]
[[** This is a subsection **]]
```

and so forth, which produce

```
\section{This is a section}
\subsection{This is a subsection}
```

and so on.

A final kind of syntactic sugar allows the programmer to visually set off definitions in both the Lisp code and the typeset documentation. For example, this:

```
;; <function read-input(filename)>
;;
;; This function reads all input from
;;  the file [[filename]] ...
```

produces:

Function read-input(filename)
This function reads all input from the file
`filename`

This definition markup is defined for functions, methods, classes, generic functions, variables, parameters, and macros.

3.4 Definitions and Chunks

LP/Lisp provides facilities for reordering the LP file to produce the documentation. There are two such facilities, definitions and chunks.

Whole sections of LP markup—documentation and code—can be defined one place in the LP document, then included elsewhere. This is useful, for example, to avoid cluttering the Lisp source file with the introduction for the documentation. In this case, the introduction might be defined at the end of the document, after all the Lisp source code, but included via a markup directive at the start of the written documentation.

The programmer makes a definition by naming a section of the LP file using the `define` tags:

```
;; <define name="intro">
;; \section{Introduction}
;;
;; This document describes LP/Lisp (Literate
;; Programming/Lisp), a Lisp ...
;; </define>
```

The definition can be included in the file elsewhere using the `insert` tag. For example,

```
;;<insert name="intro">
;;<insert name="sect1">
;;<insert name="sect2">
```

would insert three definitions in order in the output after processing their embedded LP markup, if any.

The other method of reordering output is via the *chunk* mechanism, which is very similar to other LP systems, e.g., `noweb`. Chunks are named pieces of Lisp code. Unlike definitions, they are meant to aid the flow of the written documentation, and their names are exposed to the reader. They are usually used to hide details of code at one place so as not to detract from the overall description, then to describe the code elsewhere.

This is perhaps best shown by example. Figure 2 shows the chunking mechanism in LP/Lisp.

4. THE LP/LISP PROGRAM

The LP/Lisp program is itself written in Lisp, with LP markup, and the user guide [10] was created by running the program on itself. The program was written in Allegro Common Lisp (ACL);[9] we intend to make future versions more Lisp-independent, or at least allow them to run in multiple Lisps. Some modifications will be necessary for it to run under other Lisps, primarily involving the calls to the regular expression package. The current version of the program is written in plain Lisp, that is, it is not object-oriented. The next version that is being written is fully object-oriented.

LP/Lisp is defined in its own package (`lplisp`). Thus, the user can load it via ACL's `require` function (provided that it is correctly installed to do so) or `load`, then use it with whatever other programs he or she is writing. The primary interface to LP/Lisp is the `make-latex` function, which has an extensive set of keyword parameters in addition to the file to be processed. Using the parameters, the user can:

- tell LP/Lisp to create a complete LaTeX file;
- insert a table of contents;
- set LaTeX options and packages;
- set the title, author, and date;
- set the documentation mode (implicit or explicit);
- set the directory into which to put the LaTeX file;
- tell LP/Lisp to get rid of extraneous blank lines in the code to save whitespace in the LaTeX document; and
- specify whether or not to allow LaTeX to be embedded in the Lisp code or not.

The last option may need some explanation. By default, LaTeX is allowed in Lisp code, including the partial-line comments and full-line comments embedded within functions or other Lisp expressions. However, it is entirely possible that there will be character sequences in the Lisp code itself that will cause LaTeX to have errors when it is run. For example,

[9]Franz, Incorporated; `www.franz.com`.

```
;;;
;;; <function factorial(n)>
;;;
;;; The [[factorial]] function is likely the
;;; most-used example of recursion.
;;;

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

;;; The base case is simple, just a check for
;;; $n=1$ or less:
;;;<insert-chunk name="base case">
;;; The recursive step is just $n\times (n-1)$:
;;;<insert-chunk name="recursive step">
```

Function factorial(n)

The `factorial` function is likely the most-used example of recursion.

```
[00001] (defun factorial (n)
[00002]   (cond
[00003]    <<base case>>
[00004]    (t
[00005]     <<recursive step>>
[00006]    )))
[00007]
```

The base case is simple, just a check for $n = 1$ or less:

```
[00008] <<base case>>=
[00009]     ((<= n 1) 1)
```

The recursive step is just $n \times (n - 1)$:

```
[00010] <<recursive step>>=
[00011]      (* n (factorial (1- n)))
```

Figure 2: The chunking mechanism. Left, the LP file. Right, the formatted output.

this was a problem in the LP/Lisp program itself, since it has many LaTeX commands in one form or another in some of the strings in the code. To avoid these sorts of errors, with the slight drawback of less pretty comments, the user can disallow processing within these kinds of comments.

In addition, a "shell script" interface to LP/Lisp is also provided, using the ability of Allegro Common Lisp to be invoked using the `#!` first-line convention provided by Unix-like operating systems. The `lplisp` shell script contains Lisp code to load LP/Lisp and call `make-latex` with arguments taken from the script's command line arguments.

5. CURRENT AND FUTURE WORK

We are currently in the process of revising and re-implementing LP/Lisp. As mentioned, we are changing the program to be fully object-oriented. This should make the code much cleaner and easier to modify and understand.

We are also changing the syntax of the LP markup. Our experience with the current syntax is that it is more cumbersome than need be. We do not need, in general, the full generality of something like XML. Moreover, it is usually clear, both to the reader and to the LP program itself, what the scope of a given kind of markup is, and hence, there is seldom need for ending delimiters. For example, a `<doc>` tag usually does not need the corresponding `</doc>` tag, since the documentation generally ends at the end of whatever comment it is a part of, and not before.

Furthermore, the tags are distracting for the reader. This is important, since one of our goals is to allow the comments containing the LP markup to be useful and readable to someone reading the Lisp file itself.

Consequently, in the version of LP/Lisp currently being implemented, the markup looks more like the lighter-weight markup in `WEB` and its descendants, although it differs in the details and the semantics. LP/Lisp commands are prefixed by the `@` character, for example, with a bare `@` denoting the start of a section of documentation. Where possible, we are trying to ensure that such things as section headers, function definition headers, and so forth are visually appealing and useful when reading the LP file itself.

Figure 3 shows samples of the old and new syntax for comparison.

We are also making this version of LP/Lisp much more customizable than the first version. The markup will be almost completely customizable. This includes the way commands are denoted—the user can, via setting LP/Lisp's global variables, even change the `@` character to whatever he or she wants. Sectioning can be changed as well, and the range of syntactic sugaring for fonts (or almost anything else) will be customizable, as well. For example, the defaults for typewriter font, boldface, and italics look like:

`[[typewriter]]`, `<<boldface>>`, and `__italics__`

However, the user could decide to change these to other things, for example:

`**typewriter**`, `!!boldface!!`, and
`(startital italics endital)`

The way this flexibility is implemented in the new version also has the beneficial side-effect of loosening the connection

```
;;;[[* Example *]]
;;;
;;; <function factorial(n)>
;;;
;;; The [[factorial]] function is likely the
;;; most-used example of recursion.
;;;

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

;;;<doc>
;;; The base case is simple, just a check for
;;; $n=1$ or less:
;;;<insert-chunk name="base case">
;;; The recursive step is just $n\times (n-1)$:
;;;<insert-chunk name="recursive step">
;;;</doc>
```

```
;;; @* Example
;;;
;;; @function (factorial n)
;;;
;;; The [[factorial]] function is likely the
;;; most-used example of recursion.
;;;

(defun factorial (n)
  (cond
   ;;@chunk base-case
   ((<= n 1) 1)
   ;;@end chunk
   (t
   ;;@chunk recursive-step
    (* n (factorial (1- n)))
   ;;@end chunk
   )))

;;;@
;;; The base case is simple, just a check for
;;; $n=1$ or less:
;;;@show-chunk base-case
;;; The recursive step is just $n\times (n-1)$:
;;;@show-chunk recursive-step
```

Figure 3: Comparison of LP markup in versions 1 (left) and 2 (right) of LP/Lisp.

between LP/Lisp and LaTeX. As the input file is parsed, settings of the customizable variables tell LP/Lisp what strings to look for (e.g., for a new section) and what to replace them with in the text markup output. There is nothing that restricts this translation to LaTeX; the translation could instead be to (e.g.) HTML. In order to complete the transition to another kind of text markup output, the method that outputs the text markup would need to be replaced with one for the chosen target language.

It would also not be too difficult to modify the parsing portion of LP/Lisp to allow it to parse other interpreted languages, such as Python, that could also benefit from a non-WEB-like literate programming tool.

6. CONCLUSION

LP/Lisp is a literate programming tool for Lisp. Literate programming offers many advantages for writing, maintaining, and communicating about programs. However most LP tools are targeted toward compiled languages that naturally have a step between writing and executing the code, so LP processing can be inserted in that step without seriously impacting the code writing and debugging process. Lisp programming is different, and so a different LP approach is needed. Since LP/Lisp expects LP markup and text to reside in the comments of the program, the natural Lisp programming model is unaffected, involving as it does highly-interleaved program modification and testing. Instead, LP processing is done only to produce the documentation, not the code.

The result is a very easy to use, non-intrusive means of literate programming for Lisp. LP/Lisp has been in use in our lab for about a year. This experience is currently guiding our re-implementation of the tool.

The current and future versions of LP/Lisp will be available at our Web site[10] and possibly other distribution channels (e.g., SourceForge, github, CLiki, etc.). A technical report describing LP/Lisp [10], produced by running LP/Lisp on itself, is also available there from our site.

7. ACKNOWLEDGMENTS

The author is deeply grateful to the anonymous reviewers of the submitted draft of this paper for their invaluable comments, information, and advice, and especially for pointing the author to related LP systems he was unaware of.

The author would also like to thank the members of MaineSAIL (the Maine Software Agents/Artificial Intelligence Laboratory), especially Erik Albert and James Brawn, for using and providing feedback on LP/Lisp. We are particularly grateful to Erik Albert for some modifications of LP/Lisp he carried out as a result of using it.

8. REFERENCES

[1] M. Flatt, E. Barzilay, and R. B. Findler. Scribble: Closing the book on ad hoc documentation rules. In *The 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009)*, Edinburg, Scotland, August 2009.

[2] L. Gorrie. pbook.el – Format a program listing for LaTeX. Available via the Web, `www.bluetail.com/~luke/misc/emacs/pbook.pdf` (accessed 3 October 2010), 2004.

[3] A. L. Johnson and B. C. Johnson. Literate programming using `noweb`. *Linux Journal*, pages 64–69, October 1997.

[4] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[10] `MaineSAIL.umcs.maine.edu`

[5] D. E. Knuth and S. Levy. *The CWEB System of Structured Documentation*. Addison–Wesley, Reading, MA, third edition, 2001.

[6] J. A. Krommes. FWEB: A WEB system of structured documentation for multiple languages. Web: `http://w3.pppl.gov/~krommes/fweb_toc.html`. Accessed 7 August 2010., 1998.

[7] A. Plotnick. CLWEB: A literate programming system for Common Lisp. In *Proceedings of the European Lisp Symposium*, Lisbon, Portugal, 2010.

[8] N. Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, 1994.

[9] Sun Microsystems. Javadoc tool. Available via the Web, `http://java.sun.com/j2se/javadoc` (accessed 26 July 2010)., 2010.

[10] R. M. Turner. Literate programming in Lisp (LP/Lisp). Technical Report 2010–02, Department of Computer Science, University of Maine, 5752 Neville Hall, Orono, ME 04469–5752, January 2010.