PowerLisp

Common Lisp Development Environment

Version 1.1

by Roger Corman

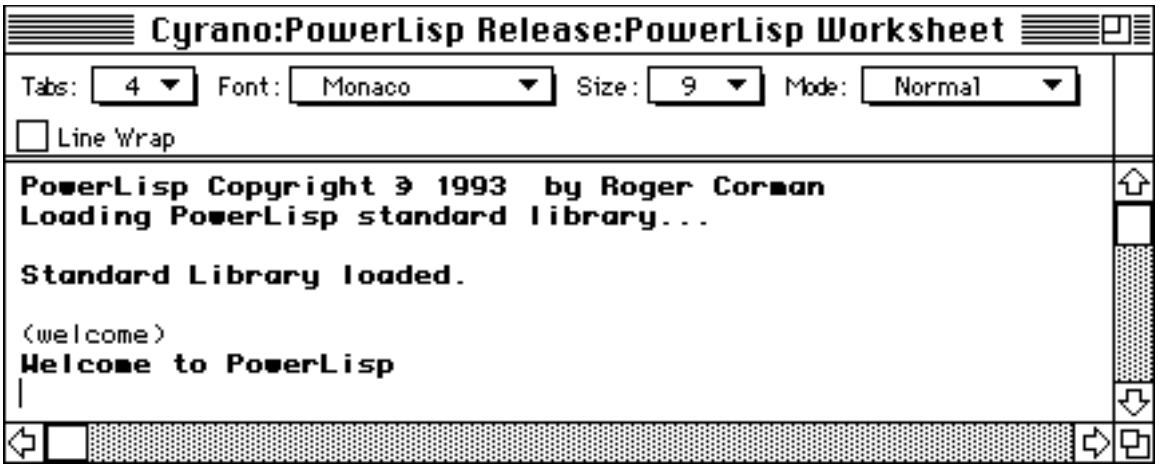
March 1, 1994

Contents

Introduction

PowerLisp 1.1 is the second public release of PowerLisp, a Common Lisp development environment for the Macintosh. It consists of a Common Lisp interpreter, native-code 680x0 compiler, 680x0 macro assembler, disassembler, incremental linker and multi-window text editor. It requires a Macintosh with at least a 68020 processor and system 7.0 or later. About 2 megabytes of RAM are required to run it, and to do much with it you need more like 5 or 6 megabytes. Like any Common Lisp system, the more memory the better.

PowerLisp has the ability to run in the background. While executing a Common Lisp program, the user may switch to another application as it continues to run.



PowerLisp is extremely fast. All compiled PowerLisp functions execute as native 680x0 instructions. The speed is comparable to other fully compiled languages.

PowerLisp programs execute in the background, while you are using the system. All editor functionality is fully available while you are compiling or otherwise executing Common Lisp programs. I do not know of another development environment on the Macintosh that can do this. Of course, you can also use a different application while your programs execute.

PowerLisp is being released as shareware. I expect to release regular updates, which will include new features and bug fixes. The frequency and scope of these will depend on the amount of interest there is in the product. I encourage everyone who finds this product useful to send me the shareware fee (see Licensing, below) and register your copy. Even if you don't elect to register it, I appreciate correspondence via e-mail or otherwise letting me know what you think of it.

The documentation is currently sparse, because of my time constraints. I intend to produce better documentation in the near future. For now the general information below will have to suffice. Please check this product out. I believe it offers extraordinary value for the price. The primary alternative for Lisp programming, Macintosh Common Lisp from Apple, is an excellent product. PowerLisp cannot compete on features or performance with Macintosh Common Lisp. It is, however, one tenth the cost ($50 as opposed to $500). I feel there may be a niche for a low-cost, small, easy-to-use product like this.

I enjoy programming with PowerLisp, and I hope you will too.

Roger Corman

Licensing

PowerLisp is distributed as shareware. I reserve the copyright to the executable application, as well as the source code to the compiler, assembler and library functions. I am asking $50 for the right to use one copy of PowerLisp. If you find this program useful, please send a check for this amount to the address below (under **Registering Your Copy of PowerLisp**). If you are a teacher, and are interested in using this for a class, please contact me. I am prepared to offer a very reasonable licensing arrangement for classes. Unlicensed copies of this system cannot be legitimately used in a class setting otherwise.

I have spent a considerable amount of time developing this program. I may elect to spend a lot more time on it, but only if licensing fees received warrant it. I do not expect a huge response to a program of this type, but each license fee I do receive will have a definite impact on the amount of time I will spend improving it.

You may share this program with others. You may not redistribute it for profit, nor make any changes to it, without the permission of the author.

PowerLisp 1.1

License Price: $50.00 US

What You Get:

- On receipt of your check, I will forward you any the most up-to-date information I have regarding bugs, new versions and features. You may also receive a beta pre-release version on floppy disk.

- You will receive at least one free major update by mail. I will send registered owners a copy of a new major revision before uploading it to on-line services.

- If you send a list of requested additional features, I will do my best to implement them as soon as possible. I will then forward you a courtesy copy.

- I will attempt to help with any problems you have and answer questions. I can only offer limited phone support. The preferred method of contacting me is via e-mail or US mail. I will answer all mail communications as soon as possible.

- If and when I decide to convert this to a commercial version (non-shareware) licensed owners of the shareware version will be guaranteed a special arrangement. This would likely be a free or at-cost license upgrade to the commercial version.

Registering Your Copy of PowerLisp

Send a check for **$50.00** US to:

Roger Corman
2124 Cummings Drive
Santa Rosa, CA 95404
USA

You may contact me by mail to the above address, or via e-mail or telephone (see below). Along with the fee, be sure to send me your address (and phone number if you don't mind). Also I would appreciate a mention of which version you have, what your system is like, and any comments you have. A wish list of product improvements would also be welcome.

America Online:
PowerLisp

Internet:
PowerLisp@aol.com

Telephone:
(707) 528-3321 (evenings and weekends)
(707) 575-4024 (days)
(707) 528-7477 (fax)

Quick Start Tutorial

This section is intended to briefly lead you through writing, running, compiling, and disassembling a small Common Lisp program with PowerLisp.

1. **Start PowerLisp by double-clicking the application icon.**
   It will take a few seconds as it loads the standard libraries. When it finishes, your worksheet will be displayed, with the blinking text cursor. The message “Ready for input” should appear in the status message area at the top of the worksheet.

2. **Turn on Lisp mode editing.**
   Although this is not the default text editing mode currently, you will typically want to use it when editing Lisp programs. Text files whose extensions end with “.lisp” automatically default to this mode. To turn it on, select **Lisp** under the **Mode** popup menu at the top of the Worksheet.

3. **Enter a PowerLisp expression.**
   Try typing:

   ```
   (list-all-packages)
   ```

   This command invokes the common lisp function which returns the packages loaded in the system. To execute it, press the **Enter** key (not the **Return** key). In PowerLisp, the **Return** key is only used for editing —to enter a new line into the text in the window. Only the **Enter** key executes anything. In this case, the entire line of text that the cursor is positioned on is read by the PowerLisp system and executed.

   Note that you may use **⌘-Return** (hold down the Command key while pressing **Return**) to simulate the Enter key if you prefer. Some keyboards do not have an **Enter** key, and so may require this method.

   After pressing **Enter** , the PowerLisp interpreter will output a list representing the packages loaded in the system. All PowerLisp output is in bold-faced text. What you type is in normal text.

4. **Create and execute a common lisp function.**
Try typing:

```
(defun print-column (x)
        "Prints the elements of a list in a column."
        (dolist (i x)
                (print i)))
```

While typing this function, use the **Return** key to end each line.

After the whole function has been entered, highlight the entire expression by clicking to the left of the opening parenthesis and dragging to the right of the ending parenthesis. After the whole expression is highlighted (all four lines) press **Enter** . The Lisp system will read and execute the expression, and then return and display the name of the defined function print-column.

Alternatively, since you are editing in Lisp mode, you may leave your text cursor positioned immediately following the last close parenthesis if the function definition. You should notice that the entire function definition is outlined. When you press **Enter**, the outlined expression is first automatically highlighted then executed.

5. **Execute the function.**
Type:

```
(print-column '(see hear taste smell touch))
```

The **print-column** function you defined will cause the list elements to be printed vertically in the worksheet (one element on each line). It is being executed by the interpreter.

6. **Get function documentation.**
Type:

```
(documentation 'print-column 'function)
```

The system will display documentation that you defined for the function.

7. **Compile the function.**
Type:

```
(compile 'print-column)
```

If this is the first time you have requested the compiler, it will take a few seconds to load the compiler and assembler. The time could take from five to twenty seconds, depending on your machine. After loading the compiler, the system will compile the function. This should take a second or less.

If the function compiles correctly, the system will print the name of the function.

8. **Execute the compiled function by repeating step 4 above.**
It is not necessary to retype this line—just go to the previous line, highlight it and press **Enter** . In PowerLisp, you should never have to retype anything!

   The system should respond as in step 4.

9. **Disassemble the function.**
Type and execute the following line:

```
(disassemble 'print-column)
```

   The system will display a dump of the machine instructions which comprise the function **print-column**. You may or may not be interested in this. Compiled PowerLisp functions always include code to check for the correct number of arguments (in this case, one).

10. **Time the function.**
You can invoke PowerLisp's high-resolution timer by executing the line:

```
(time (print-column '(see hear taste smell touch)))
```

   The function will be executed as before, and will be followed by a message regarding the amount of time elapsed during execution. You may compare this against the interpreted version by re-executing the function definition from step 3 and executing the line above again. You will see that as an interpreted function it executes slower.

11. **Save the function you have defined.**
Select the **New** command from the **File** menu. Name the new file **print-column.lisp**.

   Select **PowerLisp Worksheet** from the **Window** menu to return to the worksheet (or just click on its window). Select the function definition (from step 3) by highlighting the whole thing.

   Execute the **Copy** command via the **Edit** menu or pressing ⌘**-C**.

   Select the file **print-column.lisp** from the **Window** menu or by clicking on its window.

   Execute the **Paste** command via the **Edit** menu or pressing ⌘**-V**.

   The function definition should be displayed in the **print-column.lisp** window.

   Select **Save** from the **File** menu to save the file.

Important Notes

- You may have any number of files open. As editor memory gets filled up, temporary files may be created to store copies of the files you are editing. The number of files you have open does not affect the amount of memory available to Lisp programs (this is new with version 1.1). Actually, it has some affect but not much.

- There is no difference between the **PowerLisp Worksheet** and any other file. Every open file may act as a worksheet. Lisp output will, however, be inserted into any file which you use as a worksheet.

- If you are not using Lisp mode editing, you may enter common lisp expressions either a line at a time, pressing **Enter** after each line, or by entering a complete expression and then executing the entire thing at once. The latter is highly recommended. In the first case, if you have not entered a complete Lisp expression (perhaps not closed a list), you will see a prompt containing the number of open left parentheses in the message area at the top of the editor window.

Files in this Release

The application is called **PowerLisp 1.1**. Double click on it to launch PowerLisp.
The documentation is in a Microsoft Word format file **PowerLisp Documentation**. A folder in the PowerLisp main folder is called **Library**. It contains libraries that PowerLisp needs while running. These include:

cl.lisp	Portions of the PowerLisp standard library.
assembler.lisp	The PowerLisp assembler.
compiler.lisp	The PowerLisp 680x0 compiler.
loop.lisp	The Common Lisp Loop facility (MIT version).
backquote.lisp	Optimized backquote facility (from CLTL2, Guy Steele).
defpackage.lisp	The defpackage macro implementation.
describe.lisp	A partial implementation of the **describe** function.
format.lisp	**Format** function implementation.
graphics.lisp	Some basic graphics routines (PowerLisp specific).
structures.lisp	**Defstruct** macro implementation.

Additionally, compiled versions of these may exist along with these source files. They have the same name, with a **.fasl** extension.

The **Examples** folder contains some PowerLisp source files you may want to refer to for examples of PowerLisp functions. The file **asm-funcs.lisp** gives some examples of using the assembler. The file **eliza.lisp** is a rough version of the Eliza program from Peter Norvig's book. The examples are admittedly sparse. I intend to have some good example programs in future versions. If you have written an interesting program with PowerLisp, and you wouldn't mind having it distributed with the releases, please send it to me.

The **PowerLisp Worksheet** file is the file that normally gets loaded as the worksheet when you launch PowerLisp. If you remove or delete this file, a new one will automatically be created when you restart PowerLisp.

Compiling the Libraries

PowerLisp 1.1 is distributed with pre-compiled versions of all its libraries, so compiling the libraries yourself should not be necessary. The following section is still useful, however, if you want or need to make any changes to the library sources.

If compiled version of the libraries were not included with your release, you will want to create them yourself. Performance of compiled code is much better than interpreted code. Compiled libraries may be left out of the release to reduce disk space (and modem transfer time).

Compiling the included libraries takes a while. This is because the compiler is very slow when running in interpreted mode. Fortunately it can compile itself!
A file **compile-libraries.lisp** is included in the **Examples** folder with this release. Execute this file to compile the libraries. Give yourself as much memory as possible and be prepared to wait a while. If you like you can switch to another application and let the libraries compile in the background.

The assembler is compiled first, as this gives the best performance improvement. It starts out very slow, but speeds up as it compiles. This is because each function, as it is compiled, is dynamically linked into the run time environment, speeding future compilations. Likewise, the compiler will start to compile slowly and speed up. After the compiler is compiled, the other libraries will compile rapidly.

If compilation is successful, versions of the libraries with **.fasl** extensions should show up in your **Library** sub-folder.

The entire set of libraries will probably take from 10 to 70 minutes to compile, depending on your machine. Compiled PowerLisp files take up considerably more disk space than source files, but not necessarily more memory when loaded. This is because most of the data in the compiled file is only used for load purposes (symbol references and loader information). The size of compiled libraries is substantially reduced in version 1.1 (compared to 1.01) by eliminating much redundant symbol information.

Interactive Environment

PowerLisp is integrated with the PowerEdit text editor. The environment provides a “worksheet” approach to Common Lisp development. It is specifically modeled on the MPW environment, and also resembled the approach used by the Mathematica application.

Rather than having a window which emulates a console (e.g. the “Listener” in Macintosh Common Lisp), the worksheet approach does not emulate a console. Any number of text windows may be open, and any Common Lisp code in any open window may be executed at any time. The user typically enters a Common Lisp function or expression, highlights the expression, then presses the **Enter** key. Note that the **Enter** key is distinct from the **Return** key on the Macintosh keyboard. The **Return** key is used in the editor to insert a new line. It will not cause the PowerLisp system to interpret any text.

> **Note:** You may use ⌘-**Return** (hold down the Command key while pressing **Return)** to simulate the Enter key if you prefer. Some keyboards do not have an **Enter** key, and so may require this method.

For convenience, if no text is highlighted, the entire line of text that the text cursor is on will be interpreted whenever the **Enter** key is pressed. This allows for a usage model which is similar to a console (i.e. type a line, press **Enter**, type another line, press **Enter**). Like most Lisp consoles, until a Lisp expression is completely entered, no evaluation takes place and no output is produced. If a Lisp expression is only partially completed, the message area will display the message “**Ready for input.**” followed by the number of open left parentheses. This indicates that you are in the middle of executing an expression.

Pressing **Enter** after each line (partial expression input) should not be used when you are in the editor’s **Lisp mode**, because **Lisp mode** will sometimes cause more than just the current line to be executed.

After an entire Common Lisp expression is read, it is interpreted, and the resulting value is output at the line immediately following the line that the text cursor is on.

Since Common Lisp code can be entered from anywhere in any window, a prompt is not very useful. Output prompts also tend to get in the way of entering the next expression, as they can inadvertently get sent back as part of the next expression. Therefore, by default, PowerLisp has no prompt. You can set up a prompt by assigning the variable ***prompt*** to a function to execute. This function can output whatever prompt you want to standard output.

The worksheet approach allows you to very easily edit, execute, re-edit, and re-execute expressions without unnecessary typing. I think you will come to appreciate it as much as I do.

The front-most edit window contains a **status line**. This area, under the popup menus, is used by the system to display messages about what it is doing. **Unless the status line reads "Ready for input", you should not attempt to execute a Common Lisp expression.**

When a Common Lisp expression is being executed, you may execute editor commands, and otherwise edit files. You may also switch to another application. In this case, the Common Lisp processing will continue in the background. This is useful, for example, during a long compile. If you attempt to edit a file (with PowerEdit, the PowerLisp editor), any text output by the Common Lisp program will be directed to what was the current text insertion point at the time the Enter key was pressed (to begin the execution). I think this is generally what you want. If you are editing the same file in which the expression was executed, however, the PowerLisp output will reset the insertion point whenever it outputs text. If you are going to edit files, you probably should avoid editing the same file you are using to execute Common Lisp code.

PowerEdit Text Editor

The PowerEdit text editor does not use TextEdit (the built in text editor in the Macintosh ROM). It therefore is **not** restricted to text files of 32 kilobytes or less. In fact, it can easily handle text files over a megabyte in size. Your memory partition size determines how many files can be open. PowerEdit does not need to keep the whole file in memory (any unmodified portions are left on disk). However, the caching and memory usage are not currently as efficient as I would like. You may run out of memory if you open a lot of files.

PowerEdit, unlike TextEdit, correctly handles tabs. Tabs can be set to 1, 4 or 8 spaces for the document. Other tab settings can be added by using ResEdit to modify the Tabs popup menu resource. Each text window gets its own tab setting. Tabs get saved in the resource fork of the file, so that when the file is reopened the editor will remember the most recent setting.

The PowerEdit functions should be self-explanatory. Features include Undo, Find, Replace, Copy, Cut, Paste, Select All, and Print. The Print feature is currently pretty rough. It doesn't print anything except the text of the file (no fancy formatting).

The Window menu maintains a list of all open text files. You can use it to navigate between files when you have a lot of files open.

Scrolling

PowerEdit uses an improved (slightly different) way of scrolling than most Macintosh text editors. While you drag the scrollbar thumb, the file scrolls. Normally, in other editors, the text window does not scroll until after you release the thumb. I worked hard to get this scrolling to work this way, and am very pleased with the result. The only downside I can see is that it may be a bit sluggish on slower macs. I plan to have an option to revert to "normal" scrollbar behavior in a future version.

Font and Size Pulldowns

Each text window may have a different font and character size associated with it. This information is not currently saved with the file. The editor uses fractional widths internally to support non-monospaced font editing. Typically monospaced fonts work best for programming, however. The default font is Monaco, 9 pt. The font and font size selected are "remembered" by information in the resource fork of the file.

Document Preferences

A resource of type '**MPSR**' is added to the resource fork of any text file which is created or viewed by PowerEdit. It contains the user settings for the window position and size, the tab setting, and the font and font size. It is compatible with the method that MPW uses to save this information, so that it is convenient to alternate between PowerEdit and MPW.

Common Lisp Support in PowerEdit

PowerEdit includes some features which make it particularly useful for Common Lisp programming. For one thing, all PowerLisp interpreter output (which is sent to standard output) is printed in a bold version of the font you have selected. There is no way to enter bold text otherwise. This serves to distinguish between your input and the interpreter's output. The editor stores and remembers text style information (while editing, not when the file is saved). The PowerLisp system ignores this information, however. All text, bold or otherwise, looks the same to the interpreter.

An additional Lisp support feature involves the highlighting of parenthesized expressions. If the window is in Lisp Mode, and the text cursor is next to a parenthesized expression (a left or a right parenthesis which is balanced) the entire expression is highlighted by an outline. This is difficult to explain but relatively easy to demonstrate. Just turn on Lisp Mode from the popup menu, and enter a Common Lisp expression with several levels of parentheses.

Lisp Mode is automatically turned on by the editor for any file with a **.lisp** extension on the filename. You can explicitly turn it on or off any time from the popup menu. You may find this feature more annoying than useful. If so, turn it off. I will definitely make that an editor preference item.

Comments are no longer automatically italicized in Lisp Mode. I removed this feature because it didn't work very well and tended to make the comments difficult to edit and read.

While PowerEdit was designed to support editing with Common Lisp, it is not implemented in Common Lisp. It is written entirely in C++. While this limits the control over the editor that you have from Lisp, is allows the editor to be used in other products. I am considering releasing the editor as a stand-alone product. Let me know if you would be interested in this.

Line Wrap Mode

Line Wrap mode works better in 1.1 than it did in 1.01, but still has some bugs that show up when editing text. It is also rather slow on certain machines. You may turn on this mode, via a checkbox to the left of the status line. I find it occasionally useful to turn this mode on when **browsing** unformatted Lisp output (which may otherwise produce vary long lines). **Line Wrap** mode is not compatible with **Lisp mode**, so don't turn them on at the same time.

Recent Enhancements

The editor now uses its own heap, which is limited in size to 200k. A source file of about 80k is likely to fill it up. When the editor limit is reached, text is written to disk in a temporary file. As a result, you may edit many, large files without any effect on the space you have for your Lisp programs. I find the occasional slowdown from disk activity is preferable to the way the editor ate up memory in 1.01. Currently a temporary file for each open file is created, when the editor needs memory. These files will be automatically deleted when the program exits.

The editor has been cleaned up internally, which seems to have eliminated some spurious crashes and data loss. I still recommend that you keep good backups of your source files.

Lisp mode has been improved. As several of you noted, it seems logical that when a Lisp expression is outlined, pressing Enter should execute the entire expression, rather than just the current line. I have implemented this. To make it obvious, the whole expression gets highlighted for a fraction of a second before executing it.

Another problem several people had was getting buried in open parentheses. When you are entering expressions, and have not closed enough levels of parentheses, the system seems frozen. To make it clearer, the editor now shows the number of open parentheses in the message bar. I think this helps.

When you save a file which has bold or italicized text in it, the text attributes get stored in a resource of the file. This causes the bold text to be bold when you later load the file (it saves this attribute). You may use the edit menu commands **Bold**, **Italics** and **Plain** to control the text attributes of text in a file. These attributes are entirely ignored by the Lisp interpreter.

Editor Known Bugs

Line Wrap mode and Lisp mode are not totally compatible. I avoid using them at the same time.

Vertical bar text cursors may still get left on the screen from time to time. This is just a screen refresh issue—it does not affect the text in the file.

PowerLisp Compiler

The PowerLisp compiler is a full 680x0 native code compiler. This means that a function, once compiled, executes as direct machine instructions. This allows the compiled lisp functions to execute very fast. It is distinct from the intermediate code that some Lisp systems produce.

The compiler can be invoked on a single function, with the **compile** function, or on a source file with the **compile-file** function. The first time you call either of these functions, the compiler and assembler modules are loaded into memory. This can take from 5 to 20 seconds depending on your system. After loading, compiling is quick. A function typically takes less than one second to compile.

When the **compile-file** function is used, a binary file of machine code is produced. This file is of type **'FASL'** and typically has the extension **.fasl**. Binary files are typically about 3 times as large as the source files they originate from, but that is only because of the relatively inefficient way that all the symbol information is stored in the binary file. PowerLisp 1.1 compiled files are about 40% smaller than 1.01 compiled files (and still compatible with 1.01 files). They need to store a lot of symbol information so that all the addresses can automatically be updated correctly when the file is loaded in another system or at a later date. Once loaded, compiled code is relatively space efficient. When compiled code segments are no longer needed, the garbage collector will correctly discard them.

The entire compiler is written in Common Lisp. A couple of support functions had to be added in C++ because there is no support for packed arrays of short integers, but all the significant stuff is in the file **compiler.lisp** which is included in this release. The compiler directly generates 68000 assembler code, which it then passes off to the assembler to create the function. I could improve the compiler performance somewhat by assembling as it goes, but I have found the intermediate step useful for debugging the compiler.

The compiler generates code which generally behaves exactly like interpreted code, only faster. I typically see a 5 to 30 times speed improvement when I compile something. In terms of debugging, there are some differences between interpreted and compiled code. In at least one case, compiled code is more correct than interpreted (in the case of returning multiple values). Some special forms are not yet implemented in the compiler.

PowerLisp 1.1 Modifications

The compiler now correctly compiles **flet** special forms, structures, and special declarations. **labels** forms compile, but the generated code is the same as for **flet**.
labels works correctly when interpreted. Some bugs in the interpreter relating to these forms and function closures were identified andfixed.

The compiler now behaves in accordance with CLTL2 concerning evaluation of compiled forms. Forms are not evaluated in general, although **eval-when** can be used to force evaluation at compile time.

Compiled libraries now use the extension (by default) **.fasl** which is commonly used by other Lisp systems. Of course these files are not compatible with other Lisp systems. Compiled libraries are significantly smaller than 1.01 libraries. This is accomplished mostly by eliminating redundancy in the symbol tables. This makes loading faster. Libraries are about 30% smaller, and should still be compatible with 1.01 libraries. That is, 1.01 libraries should still load into 1.1b1, but 1.1 libraries will not be usable by 1.01.

The ‘missing function’ warning messages which so often were emitted by the compiler are now eliminated. My hope is to add a better, more useful warning facility to replace this. In the mean time it was more of a source of confusion and annoyance than a help. At any rate, if a function is missing at run time, you will get an error message (the program won’t crash).

Tail recursion is now detected and eliminated by the PowerLisp 1.1 compiler. Expressions which are written recursively but which end with a recursive call are compiled as though they were coded as iterative expressions. This eliminates much unnecessary stack usage.

PowerLisp Assembler

The assembler was designed primarily to service the compiler. It is, however, useful in its own right. It could be used as a vehicle for accessing toolbox calls and other system services which are not otherwise provided. Little support for this is included, however, in the current release.

Not all 68000 instructions are implemented, although the most common ones are. The complete source to the assembler is included with this release, in the file **assembler.lisp**. The assembler is written in Common Lisp, and all assembler instructions are implemented as macros in the assembler package. These macros automatically expand into the machine code for the instruction when expanded by the assembler in the appropriate context. This simple design allows the easy addition of assembler macros. Many sample macros can be seen in the assembler source. 68000 instructions which are not implemented could be added by anyone who wanted to take the time. I plan to expand the assembler and add a foreign function interface.

PowerLisp Disassembler

The **disassemble** function can be used to disassemble a compiled function to examine its machine code. It will disassemble functions which have been compiled by the Lisp compiler, as well as built-in functions which have been compiled by the C++ compiler. It isn't fancy, but it is pretty useful. For compiled Common Lisp functions, the disassembler is good at displaying the names of called functions (targets of **jsr** instructions). Compiled C++ functions often call functions which the disassembler does not know about, so you may get some incorrect function names. Normally you will only be disassembling compiled Common Lisp functions.

Linking and Debugging

PowerLisp features an incremental linker which immediately link in functions when they are compiled or loaded. Whenever a function is replaced by a new function, whether compiled or interpreted, all compiled branches to that function are correctly routed to the new function. This is done via a distributed jump table which is managed by the linker. Interpreted functions have a jump table entry which will cause a branch into the interpreter whenever a compiled function tries to call them. The interpreter can then, based on call stack information, determine which function was intended, and then evaluate it. If that function is later compiled, a direct jump to it replaces the interpreter branch.

Debugging facilities are rudimentary. Some non-standard functions are included which will trace the evaluation call stack or the compiled function call stack. Unfortunately there is not a single integrated function which will trace both.

While a Common Lisp program is executing, the call stack may in fact have an interpreted lisp function, which calls a compiled lisp function, which calls a compiled C++ function, which calls an interpreted Lisp function, ad nauseam. This situation is quite common, in fact. Debugging is a little easier if all the functions you are debugging are compiled, or all are interpreted.

Trace and untrace functions are useful for interpreted code. They are not of much value for compiled code. I use the non-standard functions **address** and **exec-address** a lot to get addresses which I can then examine in MacsBug.

A function called **error-stack** is included with PowerLisp 1.1. If your program aborts with an error message, you may immediately invoke this:

```
(error-stack)
```

This will print a processor dump of the top ten stack frames when the error occurred. This works better for compiled-code than for interpreted code, because all the functionson the processor stack will be interpreter functions in interpreted mode (as opposed to your functions). It still will print useful information, however.

To see the interpreter stack, you may use the function **dump-lisp-stack**. This must be invoked prior to an error occurring, however. Typically you can put it into the code of an interpreted function. When that function executes, the call to **dump-lisp-stack** will cause the top interpreter stack frames to get displayed, along with the associated lexical environments.

Memory Usage

Like most Lisp systems, PowerLisp likes to have quite a bit of memory. Garbage collection will be invoked frequently if you are short on memory, and that will cause performance to suffer.

At startup, PowerLisp sets aside enough memory to hold the application's code segments in memory, as well as some memory for operating system overhead such as windows, resources, etc. Approximately 200k bytes of RAM are set aside for the editor to hold text. About 25% of the remaining memory is given to the stack. This allows a large amount of recursion without overflowing the stack. The rest is allocated as a large non-relocatable block which is then managed by the PowerLisp memory manager.

The PowerLisp memory manager allocates about 50% of the heap to Lisp nodes. These are each 10 bytes in size, and consist of two pointers and flag and type bits. They are used to store cons cells, integers, floating point numbers, ratios, and characters. Other Lisp data types require larger blocks. All larger items and variable sized memory blocks are allocated by the other 50% of the heap. This strategy has proven to provide good performance. Fixed size cons nodes can be allocated very quickly. The garbage collector only keeps track of these nodes, or objects which are referenced by these nodes.

In a typical scenario:

PowerLisp partition size:	4096K bytes (4 megabytes)
System use:	600K
Editor heap:	200K
Stack:	900K
Nodes:	1200K (around 125,000 nodes)
Variable sized heap objects:	1200K (used by Lisp system)

Variable sized heap objects include compiled Lisp code, vectors, arrays, text editor data structures, packages and hash tables.

Memory Requirements

PowerLisp 1.1 requires at least a 2.5 megabyte partition. However, 3 megabytes is a more reasonable minimum. If you want to use the compiler, you will need at least a 3.5 megabyte partition. A larger partition is recommended, or else you will wait on the garbage collector a lot while compiling. 4 megabytes is a good size for moderate projects.

New Features in PowerLisp 1.1

A major addition is the Memory window, which can be displayed via the Memory command in the new Misc menu. It brings up a window which gives you animated displays of the PowerLisp heaps, stacks and editor heap usage. It is useful for monitoring memory usage while you are running, and noticing how much time is spent in garbage collection. When you see the Nodes and Heap bars shrinking you know garbage is being recycled.

Operating System Issues

PowerLisp runs only with Macintosh operating systems 7.0 or later. PowerLisp multitasks cooperatively with other applications, so that programs can continue running in the background while PowerLisp programs are executing. PowerLisp programs can also run in the background while you are running other applications.

PowerLisp supports the standard four Apple Events: Launch, Open, Print and Quit. It is therefore high-level event aware. It is 32-bit clean, and makes use of as much memory as you choose to give it.

PowerLisp History

I have been working on this system, off and on, for the last six years. This has never been my “real” job. I work days as a software developer of graphic arts applications. The PowerLisp system has become a passion for me, in my off hours. I began it before I had ever used a Macintosh—the first version ran on an IBM PC. It began as a highly portable C implementation, which included a subset of Common Lisp. I originally used it to debug my C programs, as an interactive scripting language.

A couple years later, after my PC had been shelved and replaced with a Mac, I resurrected the Lisp interpreter and ported it to the Macintosh. I translated it to C++, and redesigned the interpreter to be fully object-oriented internally. I also figured out the right way to do garbage collection. In overcoming the limitations of DOS 640k address space and garbage collection, the interpreter development took off. I discovered I could easily implement most features of Common Lisp in my C++ environment.

As the interpreter developed, I needed a text editor. I use MPW most of the time, so I developed a text editor, PowerEdit, which looks and acts similarly to MPW. I like the worksheet-based approach to executing commands and scripts, and modeled the interaction of the user with PowerLisp on this style. In PowerLisp you can execute Common Lisp code from any window, by highlighting the text and pressing the **Enter** key, as in MPW.

As I started to toy with the idea of actually creating a product, I decided that the system really needed a compiler for reasonable performance. I chose to build a native-code compiler entirely in Common Lisp. The source code to the compiler is included in this package. In building the compiler, I discovered and fixed a lot of problems in the system. Together with the compiler is an assembler, which assembles a subset of 680x0 assembly instructions. It is used as the second pass of the compiler, during code generation. Source code to the assembler is also included. All assembler instructions are implemented as Common Lisp macros, so it would be easy for someone to extend the assembler to include more instructions.

All the source code of PowerLisp was developed 100% by me (with the exception of the **LOOP** facility, a few of the library routines, and some of the example programs). I have not “borrowed” or licensed any technology. This is somewhat a source of pride for me, though not exactly practical. The editor is entirely my own, and does not use TextEdit, emacs, or anything else. The source code is mostly MPW C++ and Common Lisp, with a couple small routines in assembler. The Common Lisp source is included with this release.

Common Lisp Implementation

The Common Lisp implementation in this release of PowerLisp is lacking in a number of ways, which I will detail below. As you probably are aware, Common Lisp consists of a huge number of functions and data types. Rather than wait a couple more years to release this, I have tried to include the most useful features of the language. As a very rough estimate, I believe this release implements about 95% of Common Lisp as specified in the first edition of Guy Steele's ***Common Lisp: The Language***., **2nd edition.** If you don't count CLOS.

CLOS (the Common Lisp Object System) is a very important part of modern Common Lisp, and I hope to add it to PowerLisp in the future.

I would like to build a reference of what is included, because it would include a huge number of functions and features. Because of time constraints, however, I will have to base this document more on what is missing from ***Common Lisp: The Language***. While a number of things are not currently implemented, it is still a very useful system.

The following section of this document covers the PowerLisp language implementation, roughly in the order in which they are covered in Guy Steele's ***Common Lisp: The Language***, Second Edition.

In this document, I will refer to Guy Steele's ***Common Lisp: The Language***, the first edition, as **CLTL1.** The second edition of the reference will be referred to as **CLTL2**.

Data Types

Characters
Symbols
Lists
Functions
Text strings
Packages
Hash tables
Read tables

These data types are all implemented according to the language specification.

Numbers
Integers, floating-point and ratios are implemented. PowerLisp 1.1 includes complex numbers and bignums (large integers). Integers are stored in 32 bits, floating-point in 64 bits and ratios consist of two integers. Only one size of floating point number is provided. Large integers may be any size up to your memory limitations.

Arrays
Generalized arrays, bit vectors and character strings are supported.
Bit vectors and generalized arrays may be multi-dimensional, up to 7 dimensions. Character strings may only be single-dimensioned (vectors).
Arrays of specific types (packed arrays of integers, for example) are not supported, but this should largely be transparent to Common Lisp programs.

Streams
Streams are implemented, but with few variables. All streams are currently input/output.

Pathnames
Pathnames are just text strings currently—no “system independent” path name object is supported. Pathname strings can represent full path names, or partial path names relative to the default directory. They must be in Macintosh format, which uses colons rather than slashes to separate directory names.

Examples of paths:

```
"VolumeName:My Directory:My File"          ; full path
"My File"                                  ; in current directory
":My Subdirectory:My File"          ; in subdirectory of current
"::MyFile"                                        ; in parent directory (one level up)
```

Random States
The random number package is fully implemented, including the random state data type. Random state objects not currently readable, however.

Structures
Structures are implemented according to the language standard. Some of the
options are not supported yet, however. List-based structures are not supported. In PowerLisp 1.1 code which uses structures will compile correctly.

Objects
CLOS is not implemented in this version. I expect to add it in a future release. I consider it quite an important language extension, as most of my programming is object-oriented.

Scope and Extent

PowerLisp adheres to the Common Lisp specification.

New to PowerLisp 1.1 is the correct handling of **special** declarations
by both the interpreter and compiler. Local variables which are declared special should now be handled correctly. In 1.01, only variables declared via defvar or defparameter were considered special (plus some built-in variables like ***package***).

Type Specifiers

The Common Lisp type system conforms to CLTL2. No optimizations are currently done based on type declarations.

Program Structure

This area of the language is more or less complete. This includes functions, both interpreted and compiled, special forms, macros (interpreted and compiled), special variables, constants, etc. All special forms are correctly recognized.

All defining constructs (**defun**, **defmacro**, **defstruct**, **defconstant**, etc.) allow the inclusion of a documentation string. This string gets stored on the property list of the symbol, and can be accessed as specified by the language.

Compiled and interpreted functions can be freely intermixed in the call stack, and compiled functions are incrementally linked into the system as soon as they are compiled or read.

Defun and defmacro should now work correctly in non top-level contexts (this did not work in 1.01).

Predicates

Implemented.

Control Structure

Most of this chapter is implemented, with a few exceptions. Compiler macros and the **compiler-let** form are not implemented. Some of the features which are new in **CLTL2** are not implemented.

It was pointed out to me that the **do** macro in 1.01 was incorrect. In particular, it did not update the set of local variables 'in parallel' as it was supposed to do. This has been fixed in PowerLisp 1.1. Also, implementations of **psetq** and all of the multiple value handling macros are included.

Declarations

Declarations are mostly allowed but ignored by PowerLisp. In future versions of the compiler, faster code generation should be possible by paying attention to declarations. Violations of declarations are also ignored.

In PowerLisp 1.1, **special** declarations **are** significant, and are correctly interpreted and compiled.

Symbols

Implemented.

Packages

Implemented.

In PowerLisp 1.1, **defpackage** and the remaining package functions and macros have been included. There may still be some problems with the way shadowing symbols are handled, and user interaction with the shadowing facility is not supported (load time querying to resolve ambiguities).

Numbers

PowerLisp 1.1 includes most of the numeric functionality specified in CLTL2. Specifically, complex numbers have been added, large integers (unlimited size), and the trig functions. Some math functions will give an error when they encounter a large integer or complex number. The **byte** manipulation functions are not implemented, nor the **boole** function.

Integers between 0 and 500 now are cached i.e. new ones never need to be created.

```
(eq 100 100)
t

(eq 501 501)
nil
```

Characters

All chars are standard-chars. All characters are kept in a table, so never need to be created (as are integers between 0 and 500). This makes some character handling more efficient. Some character-related functions were added in 1.1.

Sequences

These functions are pretty complete. All sequence operations can be applied to lists, vectors, bit vectors and character strings. A number of missing sequence functions are now implemented in PowerLisp 1.1.

Lists

Implemented.

Hash Tables

Implemented. Hash Tables are used internally by the package system.

Arrays

Partially implemented. Arrays can be up to seven dimensions. Some key arguments to **make-array** are not implemented. Packed arrays are not implemented.

Strings

Mostly implemented. A few of these functions need to be implemented still. A number of string functions which were missing in 1.01 are now implemented in 1.1.

Structures

Structures can be defined and are correctly added to the type system. Some key arguments to **defstruct** and some slot options are not implemented yet. I intend to finish these as a precursor to CLOS support.

In PowerLisp 1.1, the **defstruct** macro has been rewritten in Common Lisp. It is included in the library **structures.lisp**. It the structure printing facility should now work correctly. Code which uses structures now compiles correctly.

The Evaluator

This is the Common Lisp interpreter. Top level run-time loop features such as **+**, **++**, **+++** and *****, ******, ******* are not implemented, as they are not really necessary in this type of environment.

Streams

Partially implemented. This area is still a little weak and is a high priority for improvement. The most important features are there, however.

Input/Output

The Lisp reader is implemented as specified, which is not easy in an event-driven environment! Read macros can be defined, and are used internally for many things (check out the standard library and compiler source code).

Options to the **format** function are partially implemented. This needs some work still. A number of other features described in this chapter are not yet implemented.

PowerLisp 1.1 has substantially improved upon PowerLisp 1.01 in this area. In particular, all the output formatting variables are supported, circular lists can be read and written, and the **format** function is much more complete.

File System Interface

As mentioned above, pathnames are just character strings currently. I intend to change this soon, so that a pathname object contains an **FSSpec** internally but can still be specified by a path string. Using strings for pathnames is compatible with Common Lisp.

Some macintosh-specific functions are available:

(**set-file-creator** my-open-file "ROSA")
(**set-file-type** my-open-file "EPSF")

These functions can be used to set the type and creator of any open file. An error is signaled if you try to call these functions on other types of streams.

File wildcard specifiers are not yet supported.

Errors

Errors are implemented as exceptions (as thrown by the **throw** special form). They are typically caught at the top level. Continuable errors are not yet implemented (for want of a debugger).

In general, all Lisp functions, both compiled and interpreted, signal errors whenever the wrong number or type of arguments is passed to them.

Miscellaneous Features

Compiler
The compiler is covered in a separate section.

Documentation
The documentation facility is fully implemented.

Debugging tools
While compiled and interpreted functions peacefully coexist at run-time, their behavior as regards debugging is significant. The macros **trace** and **untrace** are implemented for interpreted, but not compiled code. That is, you can compile a function which calls **trace** , but only the interpreted function calls will actually be traced.

The **step** function is not implemented. There is no real interactive debugger. This should be improved in a future release.

The **time** macro is implemented, and uses the Mac's Time Manager to produce microsecond timings accurate to about 20 microseconds. This is very useful for performance tuning.

The **describe** function is only partially implemented (symbols are well supported). I intend to improve it.

The **inspect** function is not implemented yet.

The **room** function can be used to determine how much memory is available. See below for more information about memory usage. Another function **gc** invokes the garbage collector. Usually you should call it before calling **room** to get an accurate result.

The **ed** function is implemented for editing files:

(**ed** filename)

causes the PowerEdit editor to open the file filename for editing. This is identical to using the Open command from the editor.

The functions **dribble** and **apropos** are not implemented.

Environmental Inquiries

These are not yet implemented.

Loop

The complete Loop facility is provided courtesy of the publicly available source code from MIT. This has been tested and run both in interpreted and compiled mode and seems to work fine. It has not been tested thoroughly, however.

Loop macros tend to expand into huge Common Lisp expressions, which execute slowly in interpreted mode but compile into pretty tight, fast code. It is like a language unto itself, and rather interesting.

The first time the system encounters a **loop** macro, it loads the loop package. This takes a few seconds. Subsequent uses of **loop** will not demonstrate this delay.

Pretty Printing

Not implemented.

CLOS

Not implemented (**CLTL2** feature). If this product finds any kind of a market I will certainly add CLOS facilities in the future. I consider it the last remaining large task.

Conditions

Not implemented (**CLTL2** feature).

Series

Not implemented (**CLTL2** feature).

Non-standard Extensions

Here are some non-standard functions and variables which are included in PowerLisp and which you may find useful.

top-level [*variable*]
This should normally be bound to the top-level read-eval-print loop.

prompt [*variable*]
If bound and non-nil, this variable should point to a function to execute during the read-eval-print loop after each iteration.

address *object* [*function*]
Returns the machine address of the lisp object that is its argument.

exec-address compiled-function [function]
Returns the machine execution address of a compiled function. If a symbol which has a compiled-function associated with it is passed, that symbol's jump table address (maintained by the incremental linker) is returned. Note that this is different from the address of the function, but normally just represents a jump instruction to the other address. This function is useful for debugging compiled code (in combination with a debugger like MacsBug).

function-definition function [function]
Returns the lambda expression of an interpreted function. As Common Lisp does not specify a standard way to retrieve the lambda expression of a function, this is a useful extension. Note that once a function is compiled, its lambda expression is discarded.

gc [function]
Explicitly invokes the garbage collector. This is more or less a standard language extension, but is not required by the standard. Use it before calling the **room** function for a more accurate estimate of space remaining.

package-hash-table *package* [*function*]
Returns the hash table used by the passed package. This is sometimes useful.

print-function *interpreted-function* [*function*]
Prints the passed function.

quit [*function*]
stop [*function*]
Exits interpreter. You probably never want to call these. Just exit the program instead (using menu Quit command).

hash-table-misses *hash-table* [*function*]
Provides statistics on hash-table effectiveness. Returns the number of times a hash-table lookup attempt has "missed" (failed).

hash-table-hits *hash-table* [*function*]
Provides statistics on hash-table effectiveness. Returns the number of times a hash-table lookup attempt has "hit" (succeeded).

set-file-type *file-stream type-string* [*function*]
Sets the Finder type for the open file. Signals an error if a stream which is not a file is passed to it.

Example:
```
(setq f (open "myfile"))
(set-file-type f "EPSF") ; sets the file's type to 'EPSF'
```

set-file-creator *file-stream creator-string* [*function*]
Sets the Finder creator for the open file. Signals an error if a stream which is not a file is passed to it.

Example:
```
(setq f (open "myfile"))
(set-file-creator f "ROSA")  ; set the file's creator to 'ROSA'
```

dump-lisp-stack [*function*]
This function prints a trace of the evaluator stack. It will only include information on evaluated function calls (not compiled functions).

%stack-trace [*function*]
This function returns a list of information on each processor stack frame. This is useful when debugging compiled functions. Evaluated function calls will show up as calls to the interpreter.

stack-trace [*variable*]
After any error, this global variable is automatically left bound to a list of stack frames that were in effect at the time of the error (as obtained with %stack-trace). This is very useful. Use the expression:

```
(error-stack)
```

after an error to see the stack trace.

error-stack [*function*]
This function may be used to print a dump of the processor stack state at the time the last error was encountered. This function can be used instead of the expression listed above.

Notes

I would like to thank my wife, Frances, for her assistance with this documentation and for her patience and encouragement.

I would also like to thank Guy Steele, whose books ***Common Lisp: The Language,*** both editions, are a constant source of assistance and amusement (in the most positive sense). PowerLisp 1.1 also includes the optimized backquote facility from the second edition, as well as some other functions from the book.

Peter Norvig's text ***Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*** taught me a lot about the language and his many sample programs were useful in debugging the interpreter and compiler. I highly recommend it to anyone learning Common Lisp.

Acknowledgements to MIT for their Loop facility source code, which I have included with this package.