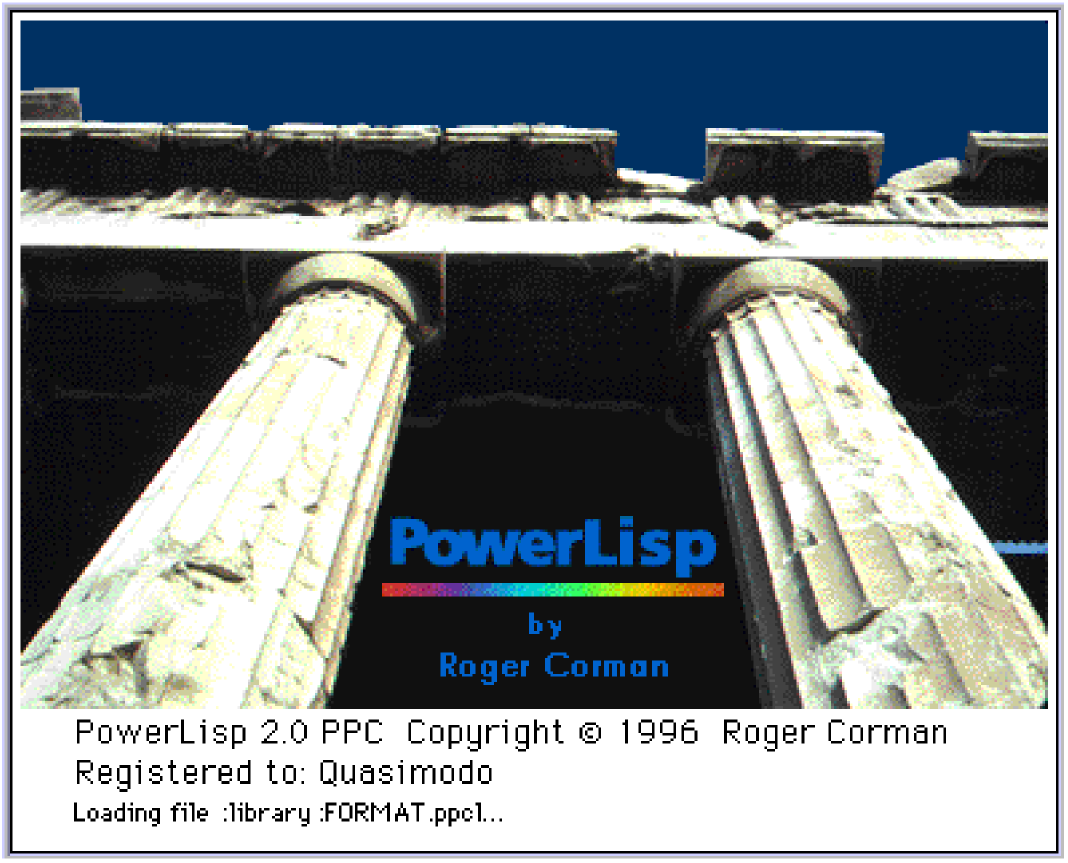


PowerLisp 2.0 PPC Copyright © 1996 Roger Corman
Registered to: Quasimodo
Loading file :library:FORMAT.ppcl...

PowerLisp

Common Lisp Development Environment

Version 2.0

by Roger Corman

May 23, 1996

Contents

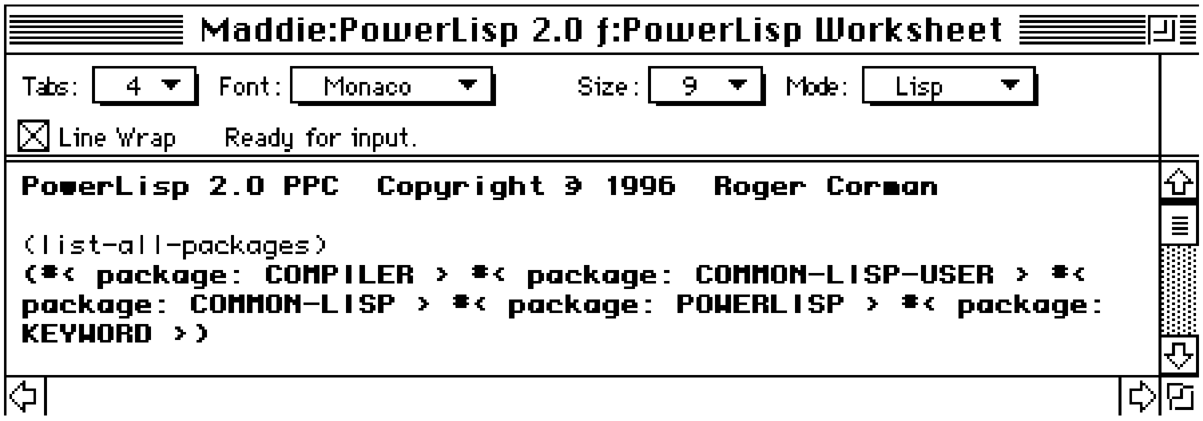
Introduction

This document is intended to provide an introduction to PowerLisp 2.0, a Common Lisp development environment for the Macintosh. PowerLisp consists of a Common Lisp interpreter, native-code 680x0 and PowerPC compilers, 680x0 and PowerPC macro assemblers, 680x0 and PowerPC disassemblers, incremental linker and multi-window text editor. It requires a Macintosh with at least a 68020 processor and system 7.0 or later, or a Power Mac. About 3 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 2.0 is a fat binary application, and will automatically run in native mode on a Power Mac, as well as continuing to run on 68k Macintoshes. PowerLisp 2.0 includes compiled libraries for both 68k Mac and Power Mac, and a PowerPC native code compiler and assembler are included.

New to PowerLisp 2.0 includes CLOS support, the Common Lisp object-oriented programming system. A number of other improvements and bug fixes have been included in PowerLisp 2.0.



Tabs: 4 Font: Monaco Size: 9 Mode: Lisp

☒ Line Wrap Ready for input.

```
PowerLisp 2.0 PPC  Copyright © 1996  Roger Corman

(list-all-packages)
(#< package: COMPILER > #< package: COMMON-LISP-USER > #<
package: COMMON-LISP > #< package: POWERLISP > #< package:
KEYWORD >)
```

PowerLisp is extremely fast. All compiled PowerLisp functions execute as native 680x0 or PowerPC 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 of a different application while your programs execute.

PowerLisp 2.0 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.

Thanks to Digital Press and Guy Steele Jr., Common Lisp documentation is now available from within PowerLisp. A hierarchical menu structure allows browsing of the full text of Common Lisp, the Language 2nd Edition. This text is distributed electronically along with PowerLisp. It is also hooked into the Common Lisp documentation function.

Please check this product out. I believe it offers extraordinary value for the price. The primary alternative for Lisp programming, Macintosh Common Lisp from Digitool, 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 $395). I feel there may be a niche for a low-cost, small, easy-to-use product like this. Also, at least for the time being, PowerLisp is the only PowerPC native common lisp that I know of. If you are using a Power Mac, and you like Lisp, you will want to take PowerLisp for a spin!

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

Roger Corman

Licensing

PowerLisp is distributed as shareware. The author of PowerLisp reserves the copyright to the executable application, as well as the source code to the compiler, assembler and library functions. A payment of $50 gives you 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, the standard license fee is $50.00 + $10.00 per student using PowerLisp. The $50.00 fee covers one permanent license, while the $10.00 fees are good for the duration of the class only. Students who wish to continue using PowerLisp may choose to register permanently at the student rate of $30.00. Unlicensed copies of this system cannot be legitimately used in a class setting otherwise.

We have spent a considerable amount of time developing this program. We may elect to spend a lot more time on it, but only if licensing fees received warrant 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.

Note: All registered users of PowerLisp 1.01, 1.1 or 1.2 automatically are registered users of PowerLisp 2.0. Contact us via e-mail or telephone (see Registering your copy of PowerLisp, below) to obtain your product registration code if you have registered and have not received your code by mail.

PowerLisp 2.0

License Price:	**$50.00 US**
Student Price:	**$30.00 US**

What You Get:

- On receipt of your check, we will forward you any the most up-to-date information we have regarding bugs, new versions and features. We will keep you informed of new versions and features as they become available. If possible, send an e-mail address, so you can be added to the soon-to-be setup PowerLisp mailing list. We will also supply you with a Product Registration Code which you may use to register your installed copy. This will disable the startup registration dialog.

- If you send a list of requested additional features, we will do our best to implement them as soon as possible.

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

- If and when we 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 (or **$30** if you are a student) to:

Roger Corman
2124 Cummings Drive
Santa Rosa, CA 95404
USA

You may contact us by mail to the above address, or via e-mail or telephone (see below). Along with the fee, be sure to send your address (and phone number if you don't mind). Also we 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. Also, if possible, please send an e-mail address where you would like PowerLisp update information to be sent.

How To Reach Us

The PowerLisp web site is at: http://www.crl.com/~rgcorman

This site will contain the most up-to-date release of PowerLisp and the most up-to-date information.

America Online:	PowerLisp
Internet E-mail:	PowerLisp@aol.com rgcorman@crl.com
Telephone:	(707) 575-4024 (voice) (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. You should regularly watch the status bar while programming within the Worksheet. It will often display messages regarding the status of the Lisp evaluator.

2. **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.

3. **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.

4. **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.

5. **Get function documentation.**
   Type:

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

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

6. **Compile the function.**
   Type:

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

   If this is the first time you have requested the compiler, it may take a few seconds to load the compiler and assembler. After loading the compiler, the system will compile the function.

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

7. **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.

8. **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).

9. **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.

10. **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. This helps prevent the number of files you have open from affecting the amount of memory available to Lisp programs.

- 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.

  New to PowerLisp 2.0: The above statement is no longer completely correct. By default, all Lisp output will now go to the Worksheet regardless of the window in which you executed the command. If you prefer the old behavior, you may enable it by turning off the **Output Always Goes To Worksheet** option in the Preferences dialog.

- 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 2.0**. Double click on it to launch PowerLisp. The documentation is in a Microsoft Word format file **PowerLisp 2.0 Documentation**. A folder in the PowerLisp main folder is called **Library**. It contains libraries that PowerLisp needs while running. These include the following lisp source files:

cl.lisp	Portions of the PowerLisp standard library .
assembler_68k.lisp	The PowerLisp 680x0 assembler .
compiler_68k.lisp	The PowerLisp 680x0 compiler.
assembler_PPC.lisp	The PowerLisp PowerPC assembler.
compiler_PPC.lisp	The PowerLisp PowerPC 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.
documentation.lisp	**Documentation** function implementation.
clos.lisp	**Clos library** (Closette implementation).
random.lisp	**Random** implementation (CMU).
startup.lisp	Run at startup, causes above libraries to load. This can never be compiled. You may add startup code to the end of this file if you wish.

Additionally, compiled versions of these may exist along with these source files. They have the same name, with a **.fasl** extension (68k binaries) or .**ppcl** (PowerPC binaries).

The **Examples** folder contains some PowerLisp source files you may want to refer to for examples of PowerLisp functions. 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 file closette-tests.lisp contains code from Art of the Metaobject Protocol, by Gregor Kiczales, Jim des Rivieres and Daniel G. Bobrow. It contains many examples of using CLOS.

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

Although compiled versions of the PowerLisp libraries should have been included with your release, you may want to recreate them yourself.

A files **compile-68k-libraries.lisp** and **compile-PPC-libraries.lisp** are included in the **Library** folder with this release. Execute the appropriate 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. Of course, you can recompile just a single library if you wish, by using the Common Lisp **compile-file** function.

PowerLisp 2.0 does not support cross-compilation i.e. you cannot compile 68k libraries on a Power Mac or PPC libraries on a 68k Macintosh.

If compilation is successful, new versions of the libraries with a **.fasl** (68k) or **.ppcl** (PPC) extensions should show up in your **Library** sub-folder.

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.

New to PowerLisp 2.0: The above statement is no longer completely correct. By default, all Lisp output will now go to the Worksheet regardless of the window in which you executed the command. If you prefer the old behavior, you may enable it by turning off the **Output Always Goes To Worksheet** option in the Preferences dialog.

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.

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.

Preferences

The amount of time PowerLisp spends executing Common Lisp programs vs. the amount of time given up to background and editing tasks is now controllable from the Preferences dialog. By selecting Most Cooperative, background tasks will run the smoothest, but Lisp programs may run slower. On older Macs, or Macs with a lot of extensions, performance of Lisp programs may be unacceptable. Least cooperative will give the least time to background and editing tasks, allowing Lisp programs to run the fastest. Background tasks will only get called once a second, which may not be acceptable. The default setting is in the middle between these extremes.

By default, all Lisp output will now go to the Worksheet regardless of the window in which you executed the command. If you prefer the old behavior, you may enable it by turning off the **Output Always Goes To Worksheet** option in the Preferences dialog.

Preferences setting must be explicitly saved, via the Save button on the Preferences dialog, or they will revert back to their original settings next time you run PowerLisp.

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. PowerEdit does not need to keep the whole file in memory (any unmodified portions are left on disk).

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. This serves to distinguish between your input and the interpreter’s output. The editor stores and remembers text style information. 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.

In Lisp Mode, when a parenthesized Lisp expression is outlined, pressing Enter will execute the entire expression, rather than just the current line. To make it obvious, the whole expression gets highlighted for a fraction of a second before executing it.

When the cursor is placed next to a lisp expression (highlighting it), double clicking will select the entire expression, which is very useful for cutting and pasting lisp expressions.

Line Wrap Mode

Line Wrap mode works much better in 2.0 than it did in any previous version 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 now compatible with **Lisp mode** (unlike with previous versions).

Formatting

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.

Lisp Mode

In the Lisp mode of the editor, when a parenthesized Lisp expression is outlined, pressing Enter will execute the entire expression, rather than just the current line. To make it obvious, the whole expression gets highlighted for a fraction of a second before executing it.

When the cursor is placed next to a lisp expression (highlighting it), double clicking will select the entire expression, which is very useful for cutting and pasting lisp expressions.

PowerLisp 2.0 Editor Enhancements

- Lisp output goes to the Worksheet by default, regardless of which window you execute a lisp expression from.
- The amount of memory allocated to the editor was increased in PowerLisp 2.0, which makes editing large files a little smoother.
- A bug that showed up on the display when dragging to off the right or left side of the window (invoking auto-scrolling) has been fixed.
- Several bugs relating to using Lisp mode and line wrap mode together have been fixed. A few screen anomalies (such as vertical bars) will still show up occasionally in line wrap mode, however.

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.

PowerLisp Compiler

68k Compiler

The PowerLisp 68k 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.

PowerPC Compiler

When running native, PowerLisp automatically loads and executes the PowerPC compiler (c**ompiler_ppc.lisp** & **compiler.ppcl**) in response to a compilation request. This compiler generates PowerPC machine code which conforms to Apple's PowerPC runtime architecture specification. When PowerLisp is running native on a Power Mac, 68k compiled lisp files (.fasl) cannot be used. Although theoretically possible to do so, by invoking the 68k emulator via the mixed mode manager, the time spent in mode switching would probably be prohibitively expensive.

Compiled PowerPC code is stored in files with a **.ppcl** extension. By running PowerLisp on a Power Mac, and compiling your lisp code to native Power Mac code, you will get the best performance possible from PowerLisp.

PowerLisp 2.0 Modifications

The special forms **flet** and **labels** are now supported by the compiler.

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.

68k

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_68k.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.

PowerPC

Not all PowerPC instructions are implemented, although the most common ones are. The complete source to the assembler is included with this release, in the file **assembler_ppc.lisp**. The above comments (regarding the 68k assembler) apply to the PowerPC assembler as well.

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.

The PowerLisp 2.0 disassembler will disassemble PowerPC code when running on a Power Mac, or 68k code when running on a 68k Macintosh.

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 to help with debugging. 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 functions on 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 300k 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 (12 bytes on Power Macs), 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:	300K
Stack:	800K
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 2.0 requires at least a 4 megabyte partition. However, 5 megabytes is a more reasonable minimum. If you want to use the compiler, you will need at least a 5 megabyte partition. A larger partition is recommended, or else you will wait on the garbage collector a lot while compiling. 6 megabytes is a good size for moderate projects.

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 is fully compatible with system 7.5.

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.**

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
Random States
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, with a few limitations.

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 using the CMU library. This results in much better randomness than the previous implementation. The random states are now readable.

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 now included with PowerLisp. See the section on CLOS for more information.

Scope and Extent

PowerLisp adheres to the Common Lisp specification.

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.

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.

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 and later, **special** declarations **are** significant, and are correctly interpreted and compiled.

Symbols

Implemented.

Packages

Implemented.

In PowerLisp 1.1 and later, **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 includes most of the numeric functionality specified in CLTL2. Some math functions will give an error when they encounter a large integer or complex number. The **byte** are now implemented in PowerLisp 2.0. The **boole** function is not implemented.

Characters

All chars are standard-chars.

Sequences

These functions are pretty complete. All sequence operations can be applied to lists, vectors, bit vectors and character strings.

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. PowerLisp 2.0 has much improved array functionality over previous versions, and most array features.

Strings

Mostly implemented. A few of these functions need to be implemented still.

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. PowerLisp 2.0 handles structures much better, more efficiently and implements more of the standard functionality than did previous versions.

The Evaluator

This is the Common Lisp interpreter. Top level run-time loop features such as **+**, **++**, **+++** and *****, ******, ******* **are** now implemented.

Streams

Partially implemented. This area is still a little weak and is a high priority for improvement. The most important features are there, however. PowerLisp 1.2 adds support for specification of read, write or read/write access when file streams are created with the **open** function.

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 2.0 supports more format options than previous versions.

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

A simple pretty printing function is now included with PowerLisp. The fully cusomizable pretty printing functionality proposed in **CLTL2** is not implemented.

CLOS

This had been added in PowerLisp 2.0. It is based on the Closette implementation in the book ***The Art of the Metaobject Protocol*** by Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. The source to this implementation is included (in the file **clos.lisp** in the library folder). This implementation of CLOS is a subset of the full CLOS specification (as described in **CLTL2**). The following limitations apply (according to Chapter 1, section 1.1 of the book):

> All the essential features of full CLOS are included: *classes*, which inherit structure and behavior from one or more classes, *instances* of classes, which are created, initialized and manipulated; *generic functions*, whose behavior depends on the classes of the arguments supplied to them; and *methods* which define the class-specific behavior and operations of generic functions. The major restrictions of the simplified dialect include:
>
> **No class redefinition.** Full CLOS allows the definition of a class to be changed; the changes are propagated to its subclasses and to extant instances. The subset does not allow classes to be redefined.
>
> **No method redefinition.** Full CLOS allows methods to be redefined, with the new definition completely replacing the old one. The subset does not allow methods to be redefined.
>
> **No forward-referenced superclasses.** Full CLOS allows classes to be references before they are defined. One class can be defined in terms of another before the second has been defined. These forward references are not permitted in the subset.
>
> **Explicit generic function definitions.** Full CLOS allows the definition of a generic function to be inferred from the method definitions. The subset requires that a generic function be explicitly introduced with a defgeneric form before any methods are defined on it.
>
> **Standard method combination only.** Full CLOS provides a powerful mechanism for user control of method combination. The subset defines only simple "demon" combination (primary, before- and after- methods).
>
> **No eql specializers.** Full CLOS allows methods to be specialized not only to classes, but also to individual objects. The subset restricts method specialization to classes.
>
> **No slots with :class allocation.** Full CLOS supports slots allocated in each instance of a class, and slots which are shared across all of them. The subset defines only per-instance slots.
>
> **Types and classes not fully integrated.** Full CLOS closely integrates Common Lisp types and CLOS classes. It is possible to define methods specialized to primitive classes (e.g. symbol) and structure classes (defined with defstruct). The subset defines classes for the primitive Common Lisp types but not for structure classes.

Minimal syntactic sugar. A number of convenience macros and special forms are not included in the subset. These include with-slots, generic function, generic-flet and generic-labels.

The book is highly recommend if you want to work with CLOS. The file clos-tests.lisp, included with PowerLisp 2.0 in the examples folder, contains most of the code samples from the book, and is useful for seeing how CLOS works.

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.

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. This function is superseded by the ANSI Common Lisp function **function-lambda-expression**, which is implemented in PowerLisp 1.1 and later, and returns the lambda expression as the first of three return values.

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.

New Features

New Features in PowerLisp 2.0

The biggest new feature in PowerLisp 2.0 is the addition of CLOS support. See the section on CLOS for details.

A Preferences dialog has been added. It lets you control how much time is given to background processes while Lisp programs are running. It also lets you choose between having output all go to the Worksheet vs. output will go to the window that last executed a Lisp command.

A registration screen has been added at startup. If you have registered, you can disable it by entering your name and product registration code in the Registration dialog. Contact Roger Corman to receive your code if you have already registered.

In PowerLisp 2.0, structures, arrays and strings are all stored more efficiently. Arrays and strings are more consistent and more closely adhere to the Common Lisp standard.

The Common Lisp variables +, ++, *, **, etc. are now properly bound during top loop execution as specified in the Common Lisp standard.

A number of other Common Lisp functions which were missing are now implemented (I am sorry to not be more specific here). Overall, PowerLisp 2.0 is much closer to the CLTL2 specification than any previous version of PowerLisp.

A serious bug which prevented PPC compiled code from working on certain Power Macs (related to the memory setup) has now been fixed. This generally caused a crash during startup on affected machines. The fix was released last summer as PowerLisp 1.2.1d. These changes are now included in PowerLisp 2.0.

New Features in PowerLisp 1.2

The most important new feature of PowerLisp 1.2 is native Power Mac execution and compilation. For 68k users, the new tagged pointer architecture allows much better handling of integers, which reduces memory requirements and improves speed. Future versions of PowerLisp should gain even more from this new architecture.

A number of bugs have been fixed, and new functions implemented. The most important bug fix has to do with floating point number comparisons, which now work correctly. My apologies to those of you who scratched your heads over this one. It was actually a one character typo in a routine called by all the comparison operators.

The **open** function pays attention for whether files are read-only, write-only, etc. This mat cause some of your code to break if it relied on PowerLisp 1.1's default to read/write permission. Read-only is not the default, as per the Common Lisp standard.

Printing of symbols and strings now correctly pays attention to the ***print-escape*** variable.

The documentation function is enhanced for version 1.2. Asking for documentation for a Common Lisp symbol now automatically brings up the appropriate text from Guy Steele's book Common Lisp the Language (thanks to Digital Press for allowing electronic distribution). Also, this book as well as PowerLisp documentation can be browsed from a hierarchical Documentation menu.

Troubleshooting

PowerLisp Startup Process

When PowerLisp launches, it first boots the kernel, which sets up all the internal heaps and data structures, including initializing symbols in the symbol tables, and loading necessary code. Once the kernel is successfully started, the file stdlib.lisp is searched for in the Library folder, which is expected to be a subfolder of the folder the application was launched from. This file normally contains code which will allow compiled lisp files to be loaded (the kernel does not have this feature built in), and then proceeds to load all the file cl.lisp. This file loads a number of other startup files as well. Compiled versions of these files are searched for first (.fasl or .ppcl extension) but if these are not found the loader will attempt to load copies of these files with a .lisp extension. Any files with a .lisp extension will run interpretively, and therefore somewhat slower, than the compiled files.

If anything goes wrong during the startup process, the system will not be left in a stable state. It may appear to perform correctly, but will likely crash on the first evaluation error (the evaluator error handler gets set up relatively late in the boot process). Generally you will get some type of error message in the Worksheet if anything went wrong during startup.

A typical problem is that not enough memory is available to load all the modules that need to be loaded. If you get a message that says **Can't copy compiled function** this actually means you are low on memory. Other low memory messages are less cryptic.

What To Do If PowerLisp Will Not Run

Some users have reported that PowerLisp freezes during startup, giving them no error message. There was a known bug in PowerLisp 1.2, fixed in 2.0, which cause a crash during startup on Macs with virtual memory turned on (including RamDoubler) or Macs with >32 megs of RAM. A development fix (PowerLisp 1.2.1d) was released to fix this problem, but some users still seem to have problems. If you have problems running PowerLisp, here are some things to try.

- **Increase the amount of memory available.** The minimum that is specified in the **Get Info** box may not be enough to start up on every Macintosh configuration. Increase this by one or two megabytes and try again. If you are then able to start PowerLisp, you can experiment with the value to get the right minimum for your system. If you are low on memory, try using RamDoubler (from Connectix) or Macintosh Virtual Memory (from the Memory Control Panel). It is not recommended that you run PowerLisp in a partition larger than the physical RAM in your system. If you try this, performance will probably not be good!

- **Remove the compiled libraries.** If increasing memory doesn't help, you can try running interpretively. You do this by moving all the .fasl (on 68k) or .ppcl (on Power Mac) files from the library folder. Put them in a different folder for safekeeping. Then restart PowerLisp. Normally this will allow PowerLisp to run, although performance will be slower.

- **Run emulated on Power Mac.** Another option, for Power Mac users, is to run PowerLisp in 68k emulated mode. You can do this by using ResEdit to edit a copy of PowerLisp 2.0. Do not modify your original. In ResEdit, open the PowerLisp 2.0 application. Delete the 'cfrg' resource. Save the application, and restart it. This will cause PowerLisp to run emulated, using the compiled .fasl 68k libraries. Performance may be better than running interpreted (above). **If you try this, please do not redistribute this modified copy of PowerLisp.**

- **Disable extensions.** If none of the above works, try rebooting your Macintosh with the shift key held down. This will disable extensions. See if this allows you to run PowerLisp.

If none of the above works, please put together a mail message containing whatever information about your system you can, as well as the version of PowerLisp you are using, and send a message to PowerLisp@aol.com. We cannot guarantee we will be able to help non-registered users, but if at all possible we will try to contact you with some advice. In any case, your messages will help us track down problems and fix them in later releases.

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 Jr., whose books ***Common Lisp: The Language,*** both editions, are a constant source of assistance and amusement (in the most positive sense). PowerLisp 2.0 also includes the optimized backquote facility from the second edition, as well as some other functions from the book.

Jim Bisso has been a valuable resource to me and helped with PowerLisp development.

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.

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

Acknowledgments to CMU for their Random facility source code, which I have included with this package.

Acknowledgments to Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow, for their excellent book ***The Art of the Metaobject Protocol***. The source code from this book is the basis for the PowerLisp 2.0 CLOS capability.

Thanks to Ken Long for his improved color icons, which are included in the released version of PowerLisp 1.2.

Thanks to MetroWerks, for building the best C++ programming environment I have ever used: CodeWarrior. The PowerLisp environment and kernel are compiled using CodeWarrior.