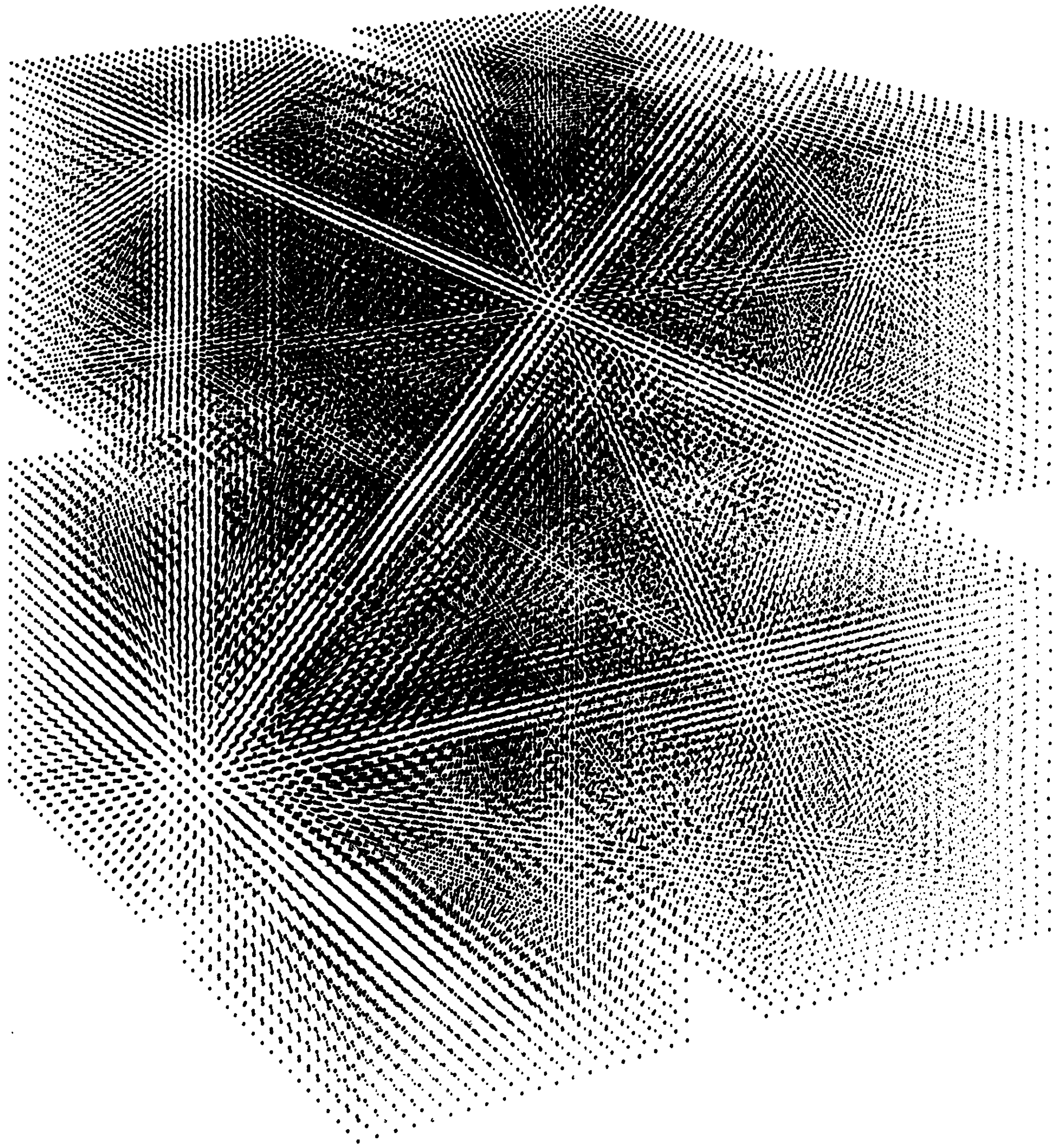
Thinking Machines Corporation

Getting Started in *Lisp



Kuszmaul

The Connection Machine System

Getting Started in *Lisp

Version 6.1
June 1991

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, June 1991

The information in this document is subject to change without notice and should not be construed as a commitment by Thinking Machines Corporation. Thinking Machines Corporation reserves the right to make changes to any products described herein to improve functioning or design. Although the information in this document has been reviewed and is believed to be reliable, Thinking Machines Corporation does not assume responsibility or liability for any errors that may appear in this document. Thinking Machines Corporation does not assume any liability arising from the application or use of any information or product described herein.

Connection Machine® is a registered trademark of Thinking Machines Corporation.
CM, CM-1, CM-2, CM-2a, and DataVault are trademarks of Thinking Machines Corporation.
Paris, and *Lisp are trademarks of Thinking Machines Corporation.
Lisp/Paris is a trademark of Thinking Machines Corporation.
VAX, ULTRIX, and VAXBI are trademarks of Digital Equipment Corporation.
Symbolics, Symbolics 3600, and Genera are trademarks of Symbolics, Inc.
Sun-4 and Sun Common Lisp are registered trademarks of Sun Microsystems, Inc.
Lucid Common Lisp is a trademark of Lucid, Inc.
UNIX is a trademark of AT&T Bell Laboratories.

Copyright © 1991 by Thinking Machines Corporation. All rights reserved.

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142–1264
(617) 876–1111

Contents

About This Manual ix
Customer Support xiii

Chapter 1 Instant *Lisp 1

1.1 Getting Started: A Simple Application 1
1.1.1 Our Project: A Cellular Automata System 2
1.1.2 *ting Up *Lisp 4

1.2 Using *Lisp 5
1.2.1 Defining an Automata Grid 5
1.2.2 Defining a Simple Automaton 7
1.2.3 Defining a More Complex Automaton 10
1.2.4 Mortal or Immortal? 17
1.2.5 Personalize Your System 18

1.3 Exiting From *Lisp 19

Chapter 2 The *Lisp Language 21

2.1 Creating and Using Parallel Variables 21
2.1.1 Creating a Pvar – **!!** 22
2.1.2 Permanent Pvars – ***defvar** 23
2.1.3 Local Pvars – ***let** 23
2.1.4 Reading, Writing, and Printing Pvar Values 23

2.2 Data Parallelism–A Different Value for Each Processor 25

2.3 Pvar Data Types 25

2.4 The Size and Shape of Pvars 27
2.4.1 Processor Grids and Configurations 27
2.4.2 ***cold–boot** 28
Configuration Variables 28
2.4.3 Processor Grids and Pvar Shapes 29
2.4.4 Pvars of Different Shapes and Sizes—VP Sets 29

2.5 Calling Parallel Functions 30

2.6 Printing Pvars in Many Ways 31

2.7 Defining *Lisp Functions and Macros 32

2.8 Compiling *Lisp Code 34
2.9 Summary: How *Lisp and Common Lisp Are Similar 35

Chapter 3 Parallel Programming Tools 37

3.1 Data Parallel Programming 37
3.2 Processor Selection Operators 38
3.2.1 Processor Selection — Doing More by Doing Less 38
3.2.2 The Processor Selection Operators of *Lisp 40
3.3 Communication Operators 41
3.3.1 Router Communication — General-Purpose Data Exchange 41
3.3.2 The Router Communication Operators of *Lisp 42
3.3.3 Grid Communication — Fast, but with Restrictions 43
3.3.4 The Grid Communication Operators of *Lisp 44
3.3.5 An Example of Grid Communication 45
3.4 Front-End/CM Communication 46
3.4.1 Funnels for Data—Global Communication Functions 47
3.4.2 Bulk Data Movement—Array/Pvar Conversions 48
3.5 Parallel Data Transformation in *Lisp 49
3.5.1 Parallel Prefixing (Scanning) 49
3.5.2 Spreading Values across the Grid 50
3.5.3 Sorting Pvar Values 51
3.6 Configuration Operators 52
3.6.1 Defining the Initial Grid Configuration — ***cold–boot** 52
3.6.2 Where Did the Extra Processors Come From? 53
3.6.3 Defining Custom Configurations — VP Sets 53
3.6.4 Default and Current VP Sets 54
3.6.5 Selecting VP Sets 55
3.6.6 Two Important VP Set Variables 56
3.7 Summary: *Lisp as a Language 57

Chapter 4 When Things Go Wrong 59

4.1 Warning Messages 59
4.2 Error Messages 60
4.2.1 Recovering from Errors 61
When You Abort, Remember to **(*warm–boot)** 61
For More Persistent Errors, Try **(*cold–boot)** 62
4.3 *Lisp Error Checking 62

4.4 Common Errors 64
4.4.1 Executing *Lisp Code without Having a CM Attached 64
4.4.2 Running Out of CM Memory 65
Running Out of Stack Memory 65
Running Out of Heap Memory 66
Tracing Stack Memory Use 67
Displaying CM Memory Use 68
4.4.3 Functions That Don't Promote Scalars to Pvars 69
4.4.4 Obscure Hardware and Software Errors 70
4.5 Using the Debugger 70
4.6 Summary: Find That Bug and Step on It! 73

Chapter 5 Declaring and Compiling *Lisp Code 75

5.1 The *Lisp Compiler 75
5.2 Compiler Options 77
5.2.1 Compiler Warning Level 77
5.2.2 Compiler Safety Level 77
5.2.3 Other Compiler Options 78
5.3 Displaying Compiled Code 79
5.4 *Lisp Data Types and Declarations 81
5.4.1 Pvar Type Specifiers — (**pvar** *typespec*) 82
5.4.2 Using Type Declarations 84
Global Type Declarations — ***proclaim** 84
Local Type Declarations — **declare** 86
In-line Type Declarations— **the** 87
5.5 Type Declaration and the Compiler 87
5.6 Example: Compiling and Timing a *Lisp Function 89
5.7 Common Compiler Warning Messages 91
5.7.1 Compiler Can't Find Type Declaration... 91
5.7.2 Compiler Can't Determine Type Returned By... 92
5.7.3 Compiler Does Not Compile Special Form... 93
5.7.4 Compiler Does Not Understand How to Compile... 94
5.7.5 Function Expected A... Pvar Argument but Got... 95
5.8 Summary: The Compiler as a Programming Tool 95

Chapter 6 *Lisp Sampler — A Scan in Four Fits **97**

6.1 Fit the First: Adding Very Large Integers 97

6.2 Fit the Second: A Segmented **news!!** Function 102

6.3 Fit the Third: Using the Router Efficiently 104

6.4 Fit the Fourth: Creating a "List" Pvar 108

6.5 Summary: There's More Than One Way to Scan a Pvar 111

Chapter 7 Going Further with *Lisp **113**

7.1 Topics We Haven't Covered 113

Creating and Using Array Pvars 113

Turning Array Pvars "Sideways" 113

Creating and Using ***defstruct** Pvars 114

Dynamically Allocating Blocks of CM Memory 114

Defining Segment Set Objects for Scanning 114

Just For Fun: Controlling The Front Panel LEDs 114

7.2 Where Do You Go from Here? 115

7.3 That's It for This Manual 115

Appendixes

Appendix A Sample *Lisp Application **119**

Appendix B A *Lisp/CM Primer **127**

B.1 Data Parallel Processing 127

B.2 Data-Parallel Processing on the CM 128

For the Curious: How It's Really Connected 129

B.3 Other Features of the CM System 130

Processor Selection 130

Processor Communication 130

Virtual Processors 131

B.4 The *Lisp Language 131
B.4.1 A Front End Language with Side Effects on the CM 131
B.4.2 Many Options for Executing Your Code 132
The *Lisp Interpreter and Compiler 132
The *Lisp Simulator 132
Running *Lisp in Batch 133
Running *Lisp under Timesharing 133

Appendix C All the Paris You Need to Know 135
C.1 Attaching to a CM: **(cm:attach)** 135
C.2 Finding Out What CM Resources Are Available 136
C.3 Finding Out if a CM is Attached: **(cm:attached)** 136
C.4 Timing *Lisp Code 137
C.5 Detaching from a CM: **(cm:detach)** 137

Appendix D Sample *Lisp Startup Sessions 139
D.1 *Lisp Hardware Startup Session 139
D.2 *Lisp Simulator Startup Session 142

Index 143
Language Index 145
Concept Index 149

About This Manual

Objectives of This Manual

*Getting Started in *Lisp* is your introduction to the *Lisp programming language. This manual provides an introduction to data parallel programming using the *Lisp language. It takes you through a sample *Lisp session, introduces you to the important data structures and parallel operations of *Lisp, and gives you the basic tools you need to start programming in the language.

Intended Audience

This manual is written for people who are relatively new to the Connection Machine® system, but who have some programming experience on other computing machines. This guide does assume a general familiarity with the design and purpose of the Connection Machine (CM) system, but the prerequisite information that you'll need is included in Appendix B of this guide. (The first chapter of the *CM System User's Guide* is also a good source for information at this level of detail, and the *CM Technical Summary* provides a deeper introduction to the CM with a more detailed description of how the CM operates.)

Because *Lisp is an extension of the Lisp programming language, familiarity with Lisp is essential. This guide assumes a general understanding of the Common Lisp dialect of Lisp, as described in *Common Lisp: The Language*, by Guy L. Steele, Jr., and assumes some programming experience in Lisp. If you're not familiar with Lisp, two good tutorial references for the language are:

> *Common Lisp: A Gentle Introduction to Symbolic Computation*, David S. Touretzky. Reading, Massachusetts: Benjamin Cummings Publishing Company, Inc., 1990
>
> *Lisp*, Patrick Henry Winston and Bethold K. P. Horn. Reading, Massachusetts: Addison-Wesley, 1984

Note: There are many small diversions and asides (much like this one) throughout the text, marked by headers such as "For the Curious" and "For Experts." These sections may be safely ignored if you wish—they provide additional information that you may find interesting or useful, and generally serve to set the record straight in places where it's necessary to sacrifice exacting technical accuracy for the sake of clarity.

Revision Information

This guide is new as of Version 6.1 of *Lisp.

Organization of This Manual

Chapter 1 Instant *Lisp

Presents a sample *Lisp session, showing you how to start up and use *Lisp, and walks you through the process of writing a simple *Lisp program.

Chapter 2 The *Lisp Language

Presents an overview of the features of *Lisp that have counterparts in Common Lisp; in other words, how Common Lisp and *Lisp are similar.

Chapter 3 Parallel Programming Tools

Presents an overview of the features of *Lisp that are specific to the CM; in other words, how Common Lisp and *Lisp differ.

Chapter 4 When Things Go Wrong

Describes the error-handling features of *Lisp, and shows you how you can use the Lisp debugger to examine and diagnose bugs in your *Lisp code.

Chapter 5 Declaring and Compiling *Lisp Code

Describes the *Lisp compiler and *Lisp type declarations, and explains why proper type declarations are essential for getting your code to compile completely.

Chapter 6 *Lisp Sampler—A Scan in Four Fits

Presents four short examples of *Lisp programs that do interesting and unusual things, with an emphasis on the way in which a particular *Lisp tool, the **scan!!** function, can be used to perform many different tasks.

Chapter 7 Going Further with *Lisp

Discusses a few topics not covered elsewhere in this guide, and presents a number of sources for further information about the *Lisp language.

Appendix A Sample *Lisp Application

Contains a complete copy of the source code for the cellular automata example of Chapter 1, along with extensions that define a generic cellular automata simulator.

Appendix B A *Lisp/CM Primer

Presents a brief introduction to the CM system and to data parallel programming, and describes how the *Lisp language is implemented on the CM.

Appendix C All the Paris You Need to Know

Presents a brief overview of a few important Paris operations that you need to know in order to use *Lisp.

Appendix D Sample *Lisp Startup Sessions

Presents sample startup sessions for both the hardware and simulator versions of *Lisp.

Related Documents

- **Lisp Dictionary.*

  This manual provides a complete dictionary-format listing of the functions, macros, and global variables available in the *Lisp language. It also includes helpful reference material in the form of a glossary of *Lisp terms and a guide to using type declarations in *Lisp.

- *CM User's Guide.*

  This document provides helpful information for new users of the Connection Machine system, and includes a chapter devoted to the use of *Lisp and Lisp/Paris on the CM.

- *Connection Machine Technical Summary*, Version 6.0.

  This document provides an overview of the software and hardware of the CM.

- *Connection Machine Parallel Instruction Set (Paris).*

  This document describes Paris, the low-level parallel programming language of the CM. *Lisp users who wish to make use of Paris calls in their code should refer to this manual.

- *Common Lisp: A Gentle Introduction to Symbolic Computation*, David S. Touretzky. Reading, Massachusetts: Benjamin Cummings Publishing Company, Inc., 1990

  This document provides a tutorial introduction to programming in Common Lisp.

- *Lisp*, Patrick Henry Winston and Berthold K. P. Horn. Reading, Massachusetts: Addison-Wesley, 1984

  This is the one of the original introductory documents for the Lisp language, and its most recent editions are compatible with the Common Lisp language standard.

- *Common Lisp: The Language,* Second Edition, Guy L. Steele Jr. Burlington, Massachusetts: Digital Press, 1990.

  The first edition of this book (1984) was the original definition of the Common Lisp language, which became the de facto industry standard for Lisp. ANSI technical committee X3J13 has been working for several years to produce an ANSI standard for Common Lisp. The second edition of *Common Lisp: The Language* contains the entire text of the first edition, augmented by extensive commentary on the changes and extensions recommended by X3J13 as of October 1989.

- *The Connection Machine,* W. Daniel Hillis.
  Cambridge, Massachusetts: MIT Press, 1985.

  This book describes the design philosophy and goals of the Connection Machine system.

Notation Conventions

The notation conventions used in this manual are the same as those used in all current *Lisp documentation.

Convention	Meaning
boldface	*Lisp language elements, such as **:keywords**, operators, and function names, when they appear embedded in text.
italics	Parameter names and placeholders in function formats.
`typewriter`	Code examples and code fragments.
> **`(user-input)`**	Interactive examples, user input.
`system-output`	Interactive examples, Lisp output.

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, a session transcript, the record of a back-trace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

To contact Thinking Machines Customer Support:

U.S. Mail: Thinking Machines Corporation
Customer Support
245 First Street
Cambridge, Massachusetts 02142–1264

Internet Electronic Mail: customer–support@think.com

uucp Electronic Mail: ames!think!customer-support

Telephone: (617) 234–4000
(617) 876–1111

For Symbolics Users Only

The Symbolics Lisp machine, when connected to the Internet network, provides a special mail facility for automatic reporting of Connection Machine system errors. When such an error occurs, simply press Ctrl–M to create a report. In the mail window that appears, the `To:` field should be addressed as follows:

`To:` customer–support@think.com

Please supplement the automatic report with any further pertinent information.

"Wilt thou show the whole wealth of thy wit in an instant?"

William Shakespeare

Chapter 1

Instant *Lisp

*Lisp (pronounced "star Lisp") is a data parallel extension of the Common Lisp programming language, designed for the Connection Machine® (CM) system.

*Lisp is a direct extension of Common Lisp, so if you've programmed in Lisp before, *Lisp programs should look very familiar to you. The data parallel extensions of *Lisp include a large number of functions and macros, and one important abstraction, the parallel variable, or *pvar* ("p-var"). As we'll see shortly, pvars are as important to *Lisp programmers as lists are to Lisp programmers.

1.1 Getting Started: A Simple Application

So that we have something specific to talk about, I'm going to pick a particular type of application and show you how to develop a *Lisp program for it. While the application I've chosen may not be identical to the projects you may have in mind, the techniques I use in developing it and the rules of thumb that I use in choosing which *Lisp features to use should carry over well to your own work.

Important: Interspersed throughout this chapter are sample *Lisp sessions that display both the *Lisp forms that you need to type, and the value(s) that *Lisp returns. In all these sessions, the lines that you'll want to type are preceded by the prompt character ">" and printed in **`bold type`**. Lines printed in `normal type` are either results displayed by *Lisp or examples of *Lisp code that you don't need to type in.

1.1.1 Our Project: A Cellular Automata System

Depending on your background and interests, you may or may not have heard of the Game of Life. Invented by John H. Conway of the University of Cambridge, the Game is played on a very large board, marked off into squares like a checkerboard. Each square can be in one of two states, traditionally called "live" and "dead." Before each game, some of the squares are made "live," usually forming a pattern of some kind. The Game itself is played by changing the state of each square according to the following rules:

- A cell's neighbors are the eight squares that surround it on the board.
- If a live square has fewer than two, or more than three live neighbors, it dies.
- If a dead square has exactly three live neighbors, it becomes live again.

All cells change their values together, in a single "step." The Game consists of a series of such steps, one after another. There are no winners and losers in the Game—the purpose of the Game is to see the kinds of patterns that arise on the board from following these three simple rules. The rules of the Game of Life cause the automaton's cells to resemble colonies of living creatures, producing patterns of "life" and "death" that ripple across the automaton's grid.

For example, Figure 1 shows a single step from one such game:

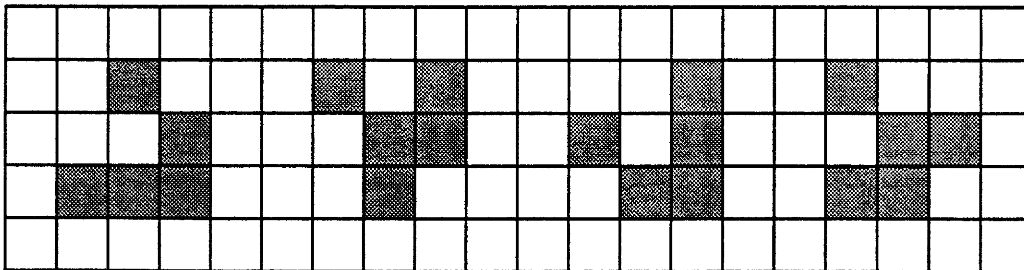


Figure 1. Picture of a single step of the Game of Life, with the "live" cells shaded

Conway's Game of Life is a simple example of a *cellular automaton*. A cellular automaton consists of a grid, or array, of elements called *cells*, each of which contains a value of some kind that determines that cell's *state*. Along with the array of cells, there is a set of rules that determine how the automaton's cells change over time. The cells change their states in a regular, step-by-step fashion, and the current state of each cell typically depends on the state values of its *neighbors*, the cells that are closest to it.

But the Game of Life is only one of a spectrum of different possible automata. Whereas the Game of Life uses two states per cell, other kinds of automata use tens, hundreds, or even thousands of states. For example, Figure 2 shows one possible "step" from such a multi-state automaton. (I've used different shading to represent different cell states.)

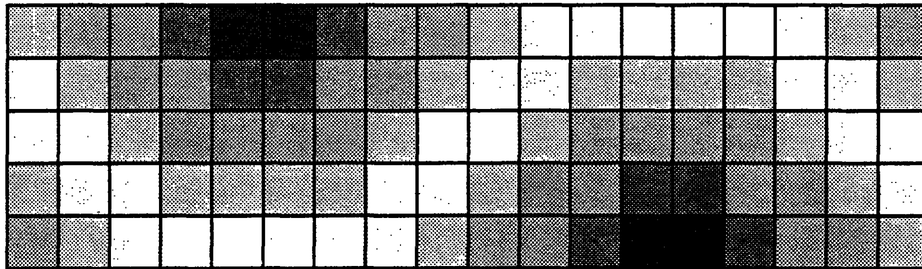


Figure 2. Picture of one step of a multi-state cellular automaton, using shading to represent the states of the cells

Such automata are used to simulate the flow of liquids or gases of varying densities, the absorption and release of molecules in chemical reactions, and even the spread of infectious diseases. Other kinds of automata use grids of different shapes and sizes; there are even automata that run on one-dimensional and three-dimensional grids.

What we'd like to produce is a simple *Lisp program that allows us to try out a number of different automata on the CM, simply by specifying the rules that describe how they work. What we need is a set of tools that will let us define and display a cellular automaton in action.

For simplicity's sake, let's assume that we'll always be working with automata that use two-dimensional grids of cells, and that the states of the cells will be represented by integers from 0 to $n-1$, with n being the total number of possible states for a cell. Thus, we need tools for setting the total number of states that each cell can have, and for defining how the value of each cell depends on the values of its neighbors. Our set of tools should also allow us to run the automaton through any required number of steps, and to display the current state of the automaton in a readable format.

Okay, now that we've got a reasonably clear idea of what we want to do, we can start up a *Lisp session and begin programming.

1.1.2 *ting Up *Lisp

At this point you have a choice to make, because *Lisp comes in two main forms. There's the standard version of *Lisp that runs on the CM hardware, and there is also a *Lisp simulator. The simulator is a Common Lisp program that runs entirely on your front-end computer, simulating the operations of an attached CM through software.

Depending on the front-end computer you are using, and where your system administrator has stored the *Lisp software, there is generally a specific command you can type that loads the software for either the CM hardware or the simulator version of *Lisp. For example, on UNIX front ends the command will typically be

```
% starlisp
```

or

```
% starlisp-simulator
```

Once you have the *Lisp software loaded, type:

```
> (*lisp)       ;;; To select the *Lisp package

> (*cold-boot) ;;; To initialize *Lisp and the CM
```

Important: The ***cold-boot** command initializes *Lisp and the CM so that you can begin typing commands. If you don't ***cold-boot**, you won't be able to execute any *Lisp code!

For Simulator Users: Among other things, the ***cold-boot** command sets the number of simulated processors that you have available. The default number of processors (32) is rather small for our purposes, so you'll want to ***cold-boot** the simulator like this:

```
> (*cold-boot :initial-dimensions '(16 16))
```

This starts you off with 256 processors, arranged in a square pattern 16 by 16 in size.

For CM Users: The ***cold-boot** command automatically attaches you to a CM, if one is available. If you get an error message such as "**the FEBI is currently in use**" or "**no sequencers are available**", it probably means that the CM is currently busy; you may have to try again later. If you're using *Lisp on real CM hardware, you can use the command **(cm:finger)** to find out what CM resources are available; see the *CM System User's Guide* for more information. If you find that the CM is busy, you can use the *Lisp simulator instead; the simulator is *always* available, regardless of the state, busy or not, of the CM.

1.2 Using *Lisp

1.2.1 Defining an Automata Grid

First, we're going to need some way of representing the grid of cells for the automaton. In a program written for a serial computer, the grid would be a two-dimensional array of integers. On the CM, however, we can take advantage of the parallel nature of the machine by storing the state value for each cell in the memory of a different CM processor.

To do this, we'll define a parallel variable, or *pvar*, to keep track of the grid:

```
> (*defvar *automata-grid* 0)
*AUTOMATA-GRID*
```

By default, when you ***cold-boot** *Lisp the CM processors are automatically arranged in a two-dimensional grid that is as close to being square as the number of available processors permits.

Let's print out a portion of this grid:

```
> (ppp *automata-grid* :mode :grid :end '(8 5))

0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

The **ppp** function (also known as **pretty-print-pvar**) prints out the values of a pvar in whatever format you specify. Here we're using it to display the state values of our **automata-grid** as a grid of numbers.

We'll want to do this quite often, so let's define a function to display the grid:

```
> (defun view (&optional (width 8) (height 5))
    (ppp *automata-grid* :mode :grid :end (list width height)))
```

Each value displayed by **view** comes from the memory of a single CM processor. We can read and write each value individually, much as we can read and write the individual elements of an array.

For example:

```
> (defun read-cell (x y)
    (pref *automata-grid* (grid x y)))
> (read-cell 5 1)
0
```

The *Lisp operation **pref** reads a value from a pvar, given its location within the CM. To specify the location I've used **grid**, a *Lisp helper function that lets you describe the location of a particular value by its grid (*x* and *y*) coordinates. Applying ***setf** to **pref** (by analogy with Common Lisp's **setf**) changes the value of the pvar at the specified location:

```
> (defun set-cell (x y newvalue)
    (*setf (pref *automata-grid* (grid x y)) newvalue))
> (set-cell 5 1 7)
> (read-cell 5 1)
7
> (view 8 3)
0 0 0 0 0 0 0 0
0 0 0 0 0 7 0 0
0 0 0 0 0 0 0 0
```

It would be a nuisance to have to set the state value of each cell in the grid individually, so let's define a function that will let us set the state of all the cells at once:

```
> (defun set-grid (newvalue)
    (*set *automata-grid* newvalue))
> (set-grid 7)               ;;; Set cells to same value
> (view 8 3)
7 7 7 7 7 7 7 7
7 7 7 7 7 7 7 7
7 7 7 7 7 7 7 7

> (set-grid (random!! 10)) ;;; Set cells to random values
> (view 8 3)
1 3 0 5 3 1 9 3
8 5 6 6 0 0 4 6
4 4 5 9 6 7 5 6
```

Notice that the **random!!** function calculates a *different* random value for each cell, rather than choosing one random value for all of the cells. This is an important feature of *Lisp: *data parallelism*. *Lisp operations such as **random!!** cause every processor to perform the same operation on potentially different data; each processor can produce a different result.

1.2.2 Defining a Simple Automaton

Now we can start thinking about how to define the rules for an automaton. The simplest type of automaton is one in which each cell's state depends only on its previous state—that is, where neighboring cells have no effect on one another.

As an example, let's define an automaton that obeys the following rules:

- Each cell can have any state value from 0 to 9.
- If a cell's state is even, divide its state value by two.
- If a cell's state is odd, add 1 to its state value and multiply by 2.

I'm not just picking these rules out of thin air. We'll see the effect they have in a moment.

In *Lisp, we can easily write a function that implements these rules. For example, here's a function that does the calculations for a single step of the automaton:

```
> (defun one-step ()
    (if!! (evenp!! *automata-grid*)
       (floor!! *automata-grid* 2)
       (*!! (1+!! *automata-grid*) 2)))
```

Here, I'm using the *Lisp operator **if!!**, which works much like **if** in Common Lisp; it evaluates a test expression for all the CM processors, and then performs one *Lisp operation for all processors where the test was true (not **nil**), and another operation for all processors where it was false (**nil**).

In the **one-step** function we use **evenp!!** as the test to find those processors whose cell values are even. For these processors, the value of ***automata-grid*** is divided by 2. For all the remaining processors, the value of ***automata-grid*** is incremented by 1, and then multiplied by 2, using the parallel operators **1+!!** and ***!!** respectively.

For the Curious: Why am I using **floor!!** rather than **/!!**? In *Lisp **/!!** is defined to always return a floating-point result, because *Lisp doesn't include "rational number" pvars. To get an integer result instead, I use **floor!!**.

Each calculation used by **one-step** takes place simultaneously in all processors. Even more important, the value of ***automata-grid*** is not changed by the **one-step** function. **if!!** returns a pvar whose values depend on the results obtained from the two **if!!** branches. The value of this pvar for each processor is the value of the branch that processor executed.

This means that all we need to do is type **(set–grid (one–step))** to calculate the new value for each cell and update the entire grid. But there's an important question that we need to answer first: What happens if a cell's value exceeds the total number of states allowed? Cells are only permitted values between 0 and 9, but **one–step** can return a value for a cell that is greater than 9.

This suggests a simple answer: We can use the *Lisp function **mod!!** to redefine **set–grid** so that it will automatically wrap values greater than 9 back into the range 0 through 9:

```
> (defvar *total-number-of-states* 10)

> (defun set-grid (newvalue)
   (*set *automata-grid*
      (mod!! newvalue *total-number-of-states*)))

> (set-grid 27)
> (view 8 3)
7 7 7 7 7 7 7 7
7 7 7 7 7 7 7 7
7 7 7 7 7 7 7 7

> (set-grid (random!! 27))
> (view 8 3)
7 2 0 7 3 8 6 4
1 0 8 3 2 4 5 2
1 9 7 0 2 3 1 3
```

This last example is a useful tool. We'll want the ability to set every cell in the grid to a random state. Let's write a function that does this, and write it so that it automatically chooses random values in the range from 0 up to one less than the current number of states:

```
> (defun random-grid ()
   (set-grid (random!! *total-number-of-states*)))
```

And while we're defining things, let's create a function that will run the automaton through any specified number of steps:

```
> (defun run (&optional (n 1))
   (dotimes (i n)
      (set-grid (one-step))))
```

Now we can run the automata and see how it works. Let's randomize the grid and run the automaton through 1, 5, and 50 cumulative steps:

```
> (random-grid)
> (view 8 3)
5 3 2 4 6 7 6 3
0 9 5 2 1 2 9 7
7 5 3 8 6 8 7 8

> (run)
> (view 8 3)
2 8 1 2 3 6 3 8
0 0 2 1 4 1 0 6
6 2 8 4 3 4 6 4

> (run 4)
> (view 8 3)
1 4 4 1 1 2 1 4
0 0 1 4 2 4 0 2
2 1 4 2 1 2 2 2

> (run 45)
> (view 8 3)
1 4 4 1 1 2 1 4
0 0 1 4 2 4 0 2
2 1 4 2 1 2 2 2
```

We don't have to run this automaton for many more steps to realize that it's in a steady loop; all 0 cells will remain 0, and all other cells will run through the series 4–2–1 over and over.

I chose the original rules with this result in mind; it's one of the most common ending conditions of a cellular automaton. After a certain number of loops, a cellular automaton typically settles down into one of a few basic patterns of activity:

- It can become moribund, with no cells ever changing.
- It can become trapped in a steady, repeating pattern.
- It can become chaotic, displaying no readily apparent pattern.
- It can alternate irregularly between steady patterns and chaotic activity.

Let's define another automaton, this time one that's a little more interesting (that is, one of the latter two varieties).

1.2.3 Defining a More Complex Automaton

This time we'll create an automaton in which neighboring cells *do* influence one another. The automaton I have in mind is a variation on Conway's Game of Life, called "9 Life." As before, each cell can have any one of ten states, 0 through 9, but the rules are somewhat more detailed:

- For each cell in the automaton, get the state values of the cell's neighbors, and count the number of neighbors that have non-zero values.
- If the number of non-zero neighbors is either less than 1, or greater than 3, subtract 1 from the cell's value (unless it is already zero, in which case it remains zero).
- If the number of non-zero neighbors is either 2 or 3, add 1 to the cell's value.
- Otherwise leave the cell unchanged.

Rest assured, there is method to this madness, as we'll see when we run this automaton.

Of course, I also have to define what I mean by "neighbors". There are two well-known types of neighborhoods that are used in two-dimensional cellular automata, the Von Neumann neighborhood and the Moore neighborhood (see Figure 3).

The Von Neumann neighborhood is the four cells immediately adjoining a given cell; in other words, its neighbors to the north, east, south, and west. The Moore neighborhood adds the four diagonal neighbors as well, for a total of eight cells.

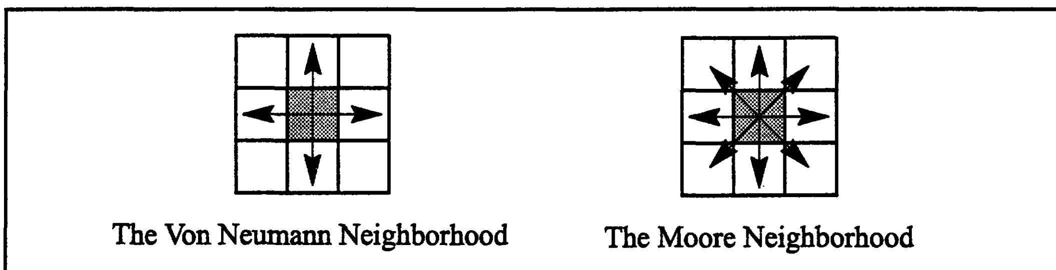


Figure 3. Two important cellular automaton neighborhoods

The Game of Life happens to use the Moore neighborhood, but we'll design our tools so that we can choose either neighborhood when we run the 9 Life automaton—that way we can see the differences between the two.

No matter which neighborhood we choose, however, we'll need a function that will cause each cell to check the values of its neighbors, make a count of the ones that are currently positive, and update its own value.

If we were to write such a function for an ordinary serial computer, it would most likely consist of a loop that steps through the automaton cell by cell, doing the count for each cell individually. A serial **one-step** function would look something like:

```
(defun serial-one-step (array-x-size array-y-size)
   (dotimes (x array-x-size)
      (dotimes (y array-y-size)
       (let ((neighbors (list (get-cell (- x 1) y)
                              (get-cell (+ x 1) y)
                              (get-cell x (- y 1))
                              (get-cell x (+ y 1))
                              ... )))
         (store-new-cell-value x y
            (new-value neighbors)))))
   (update-grid))
```

Here, **get-cell** gets the value of a cell from the automata grid, **new-value** calculates the new value for a cell based on the values of its neighbors, **store-new-cell-value** stores the new value of the cell in a temporary array, and **update-grid** updates the grid by storing the new value for each cell into the automata grid.

There are three important things to notice about this function:

- The function must explicitly refer to the actual shape of the grid, as described by the arguments **array-x-size** and **array-y-size**.
- The function includes a double **dotimes** loop to step through all the cells.
- The function must temporarily store the new value for each cell, and then update the values for all the cells after the loop has been completed, so that the new values of cells reached earlier in the loop don't influence the calculations for cells reached later in the loop.

Things are much simpler on the CM. Remember that each cell of the automaton is stored in the memory of a different CM processor. We can simply have the processor associated with each cell find the state values of the cell's neighbors, count the ones that are positive, and then update the cell accordingly. All the cells will update simultaneously, eliminating the need for any looping as well as the need to temporarily store the new value for each cell outside the grid. Before we can write the function that does this, however, we'll need to look at how we can use *Lisp operations to cause the CM processors to exchange values.

There are a number of *Lisp operators that cause the CM processors to exchange values with each other. What we want is a operator that will instruct each cell in the automaton to get the value of a single neighboring cell (for example, one to the north, one to the east, etc.).

In *Lisp, the tool we want is **news!!**. For example:

```
> (random-grid)
> (view 8 3)
2 3 1 0 2 7 2 4     ;;; Dimension  0 runs "across"
4 5 6 5 4 6 8 0     ;;; Dimension  1 runs "down"
6 8 4 3 8 1 7 9

> (set-grid (news!! *automata-grid* -1 0))
> (view 8 3)
0 2 3 1 0 2 7 2
1 4 5 6 5 4 6 8
0 6 8 4 3 8 1 7
```

In this example, I've used **news!!** to shift the entire contents of the ***automata-grid*** by one cell. The grid coordinates given to **news!!**, (–1, 0), are interpreted by each processor as being relative to that processor's position in the grid. So the coordinates (–1, 0) cause each processor to get a value from the processor that is one step to the "west" across the grid. Each processor gets the cell value of its west neighbor, and stores that value in its own cell.

For the Curious: You may be wondering where the numbers in the left-most column came from. A useful property of **news!!** is that it "wraps" automatically at the edges of the grid. The cells on each edge automatically wrap to their counterparts on the opposite edge, so there's no need to do anything special at the edges of the grid. This also means that we don't have to worry about the actual size of the grid. Our code will run unchanged on CMs of any processor size.

Who Are the People in YOUR Neighborhood?

Let's use **news!!** to define a set of functions that will cause each cell to count the number of its non-zero neighbors according to the current neighborhood we've chosen.

We'll want to be able to switch neighborhoods, so let's define a global variable that will determine the neighborhood that we want these functions to use.

```
> (defvar *neighborhood* :neumann)
```

Now for the counting functions themselves. There are many ways to write them, of course, but what we want is something simple and readable. Here's a function that will get the sum of neighbors in the Von Neumann neighborhood:

```
> (defun neumann-count (grid)
    (+!! (news!! grid  0 -1)  ;; north
         (news!! grid  0  1)  ;; south
         (news!! grid -1  0)  ;; west
         (news!! grid  1  0)  ;; east
         ))
```

and here's one that does the same thing for the Moore neighborhood.

```
> (defun moore-count (grid)
    (+!! (news!! grid  0 -1)  ;; north
         (news!! grid  0  1)  ;; south
         (news!! grid -1  0)  ;; west
         (news!! grid  1  0)  ;; east
         (news!! grid -1 -1)  ;; northwest
         (news!! grid -1  1)  ;; southwest
         (news!! grid  1 -1)  ;; northeast
         (news!! grid  1  1)  ;; southeast
         ))
```

It will be really convenient if we can call a single function and have it choose the right helper function for our current needs. Here's the function we'll use:

```
> (defun neighbor-count ()
   (*let ((grid (signum!! *automata-grid*)))
      (ecase *neighborhood*
         (:moore (moore-count grid))
         (:neumann (neumann-count grid)))))
```

Here we've used ***let**, the *Lisp version of **let**, to define a local pvar whose value is a copy of the ***automata-grid*** with all non-zero cells set to 1. When we pass this grid to the neighbor-counting functions, we obtain just a count of the number of non-zero neighbors.

For the Curious: The *Lisp function **signum!!** is the parallel version of Common Lisp's **signum**; the **signum!!** function takes a numeric pvar argument, and returns a pvar that contains 1, 0, or –1 for each processor, depending on whether the argument pvar was positive, zero, or negative for that processor.

One Small Step...for 9 Life

Having these neighbor-counting functions, we can redefine the **one-step** function for the 9 Life automata as:

```
> (defun one-step ()
   (*let ((count (neighbor-count)))
    (cond!!
      ; When count is < 1 or > 3, subtract 1 if not zero.
      ((or!! (<!! count 1) (>!! count 3))
       (if!! (zerop!! *automata-grid*)
            *automata-grid*
            (1-!! *automata-grid*)))

      ; When count is 2 or 3, add 1
      ((<=!! 2 count 3) (1+!! *automata-grid*))

      ; Otherwise, leave cells unchanged
      (t *automata-grid*))))
```

In this function we're using another *Lisp conditional operator, **cond!!**. Similar to Common Lisp's **cond** form, **cond!!** evaluates multiple tests and executes branches of code in separate groups of processors.

As with **if!!**, **cond!!** returns a pvar whose value depends on the execution of its branches. The value of a **cond!!** form for each processor is simply the value of the **cond!!** clause that processor executed. This is handy, because we want to set the value of ***automata-grid*** based on the results of a number of parallel tests, but we don't want to have to write a separate **(set-grid (if!! ...))** expression for each test.

Now in order to see the effects of this automaton we'll want to set up a more orderly test case than a random grid. This pair of functions will initialize the grid with a simple pattern:

```
> (defun set-cells (cell-list value)
    (dolist (cell cell-list)
     (set-cell (car cell) (cadr cell) value)))

> (defun init ()
    (set-grid 0)
    (set-cells '((2 2) (3 1) (3 2) (3 3) (4 1))
               1)
    (view))
```

And let's add a function to our toolbox that will let us run the automaton through a number of steps, and then automatically display the current state of the automaton. This is easy—we'll just reuse the **run** and **view** functions we defined earlier:

```
> (defun view-step (&optional (n 1))
    (run n)
    (view))
```

Let's run this new automaton through 1, 5, and 50 cumulative steps:

```
> (init)
0 0 0 0 0 0 0 0
0 0 0 1 1 0 0 0
0 0 1 1 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0

> (view-step)
0 0 0 0 0 0 0 0
0 0 1 2 1 0 0 0
0 0 1 2 1 0 0 0
0 0 1 1 0 0 0 0
0 0 0 0 0 0 0 0

> (view-step 4)
0 0 0 0 0 0 0 0
0 0 5 6 5 0 0 0
0 0 5 0 5 0 0 0
0 0 5 5 4 0 0 0
0 0 0 0 0 0 0 0

> (view-step 45)
0 0 0 0 0 0 0 0
0 0 9 7 2 0 0 0
0 0 4 0 6 0 0 0
0 0 3 4 8 0 0 0
0 0 0 0 0 0 0 0
```

Strange, isn't it? The automaton seems trapped in a square pattern of numbers, determined by the shape of the original pattern of 1's. The cells within the square continue to change in a manner that is not readily predictable, but this pocket of non-zero values can't seem to spread beyond the edges of the square.

We're running the automaton with a Von Neumann neighborhood, so that each cell has a limited number of neighbors. What happens if we use the Moore neighborhood?

```
> (setq *neighborhood* :moore)
:MOORE
```

Here's the results for runs of 1, 5, and 50 steps:

```
> (init)
0 0 0 0 0 0 0 0
0 0 0 1 1 0 0 0
0 0 1 1 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0

> (view-step)
0 0 0 1 1 0 0 0
0 0 1 2 2 0 0 0
0 0 2 0 0 0 0 0
0 0 1 2 1 0 0 0
0 0 0 0 0 0 0 0

> (view-step 4)
0 1 0 0 0 0 1 0
1 0 2 2 4 0 1 0
1 0 0 0 0 2 1 2
1 0 3 4 0 0 0 1
1 0 3 4 0 0 0 1

> (view-step 45)
0 0 0 0 3 0 0 0
4 6 6 1 0 0 0 0
0 0 0 0 0 0 4 3
3 3 8 1 0 0 5 1
0 0 0 0 2 1 0 1
```

As you can see, the eight-fold pathways of the Moore neighborhood allow the original pattern to spread much further across the grid.

For Speed Freaks: By the way, if the **(view-step 45)** calls in the above examples seem slow to you, remember that you're running these examples in an interpreted form, which will by definition not run at the maximum speed of your machine. You can compile your *Lisp code for speed, turning it into efficient Lisp/Paris code—we'll see how to do this later on.

1.2.4 Mortal or Immortal?

An interesting question to ask is: Does the pattern die out for either neighborhood? That is, does the automaton ever reach a state at which every cell is in the zero state, so that no further change is possible?

I'll leave that for you to answer, by exploring further the two versions of the automaton. To try either one, remember to choose a ***neighborhood***, either **:neumann** or **:moore**, and to call the **init** function to restore the original pattern. Then **view-step** yourself into oblivion.

There's one more tool that you'll need to use to check whether the grid is truly dead. As you know, **view** shows only a portion of the grid. But the state values produced by the Moore version of this automaton can ripple across the grid, into cells that we can't see using the default **width** and **height** arguments to **view**.

Rather than print out the entire grid to make sure there are no live cells left, you can use a simple *Lisp function to do the test for you. It's called ***sum**:

```
> (*sum (signum!! *automata-grid*))
208
```

The ***sum** function returns the sum of all the values of a pvar. In this example, we've once again made a copy of the ***automata-grid*** in which every non-zero cell is replaced by 1, and then used ***sum** to take the sum of this grid's values. This gives a count of the number of live cells in ***automata-grid***. When the count reaches zero, the automaton is well and truly dead. So a simple test for the dead state is:

```
> (defun deadp () (zerop (*sum (signum!! *automata-grid*))))
> (deadp)
NIL
```

Notice that we're using the Common Lisp **zerop** test here, not a parallel test, because ***sum** returns a scalar value, not a pvar. When **deadp** returns **t**, you'll know that every cell in the grid—including the ones that you can't see with **view**—has been zeroed out.

Try running both versions of the 9 Life automaton for a few hundred steps, and see if either one dies out. Then try modifying the rules for the automaton, or the initial pattern, and see what happens as a result. See if you make the pattern die out faster, or cause it to fall into a fixed state forever.

1.2.5 Personalize Your System

If you've been typing things in all along, and saving the **defun** and **defvar** forms into a file as you work, you now have a fairly sizable amount of *Lisp code that implements a limited but generic cellular automata simulator. What do you do with it now? Personalize it, of course. Play around with the code yourself, and modify things to suit your own taste. Enhance the system with your own ideas; there are many ways in which it can be improved.

For example, the way things are right now you must redefine the **one–step** function every time you want to try a different automaton. A useful way to extend this automata simulator would be to write operations that let you define many different kinds of automata by name, and to call up any named automaton for use automatically.

In an appendix to this document, I've outlined just such a system for your perusal. The appendix also includes a commented copy of all of the *Lisp code used in this chapter. There is also a copy of this code on-line in the *Lisp software directory, in the file

```
/cm/starlisp/interpreter/f6100/cellular-automata-example.lisp
```

Check with your system administrator or applications engineer if you need help locating this file.

1.3 Exiting From *Lisp

At this point, let's assume you're ready to call it quits for now, and want to exit from *Lisp.

If you're attached to a CM, type

```
> (cm:detach)      ;; Detach any attached CM
```

to release it.

You can then type the Lisp exit command that is appropriate for your system, as described in the *CM User's Guide*. For example, Lucid users should type either **(lcl:quit)** or **(sys:quit)**. Symbolics users don't need to "exit" from Lisp, and can simply type **(*lisp)** to deselect the *Lisp software package.

Note: If you detach, and then decide that you want to do some more work with *Lisp, simply type

```
> (*cold-boot)
```

to reattach and re-initialize *Lisp.

Remember: until you call ***cold-boot** to attach a CM and initialize *Lisp, you won't be able to execute any *Lisp code!

"But still the heart doth need a language, still
Doth the old instinct bring back the old names."
Samuel Taylor Coleridge

Chapter 2

The *Lisp Language

In Chapter 1 we looked at starting up *Lisp and writing simple applications in the language. In this chapter and the next, we'll look at *Lisp as a language and see what features it contains. This chapter describes the features of *Lisp that resemble existing features of Common Lisp—in other words, how the two languages are similar.

Briefly, in this chapter we'll be looking at:

- pvars, the fundamental parallel data structure of *Lisp
- data parallelism
- the *Lisp parallel equivalents of Common Lisp functions
- the parallel data types of *Lisp
- defining and compiling your own parallel functions

2.1 Creating and Using Parallel Variables

The basic parallel data structure in *Lisp is the parallel variable, or *pvar*. A pvar is a variable that has a separate value for each processor in the CM. Each processor can independently use and modify its own value for a pvar.

The values of a pvar can be any one of a number of front-end data types, including numbers, characters, arrays, and structures. I'll often speak of these data types as being *scalar* data objects to distinguish them from the *parallel* pvars stored on the CM.

2.1.1 Creating a Pvar – !!

The simplest operation of *Lisp is !! (pronounced "bang-bang"), which takes a single scalar argument and returns a pvar with that value in every processor.

For example, call the function !! with your age. In my case, this looks like:

```
> (!! 24)
#<FIELD-Pvar 1-5 *DEFAULT-VP-SET* (64 128)>
```

Presto! You've just told your age to several thousand processors. You've also created a pvar (displayed as "#<FIELD-Pvar...>"). Each processor in the CM has reserved a region of space in its memory to hold the value you specified. The sum total of all those regions of reserved memory within the CM is the pvar.

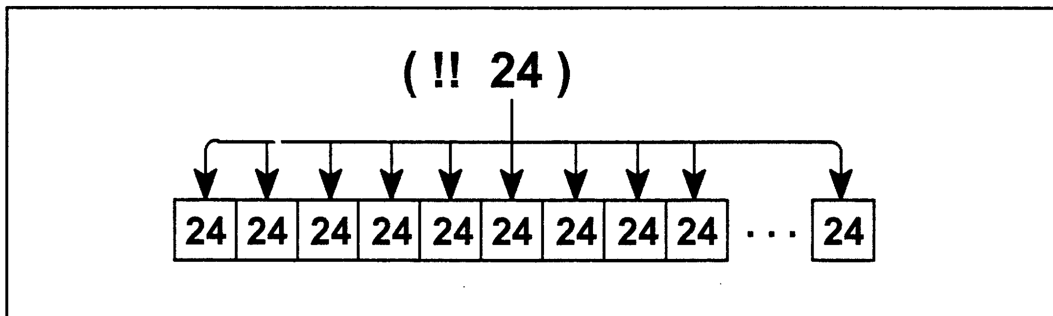


Figure 4. The expression **(!! 24)** distributes a scalar value (24) to all processors

This process of creating space for a pvar in the memory of all processors on the CM is referred to as *allocating* a pvar, and the corresponding operation of releasing that space is referred to as *deallocating* a pvar. (This is not because *Lisp programmers like long words for simple things; allocating pvars is a little more involved than creating variables in Common Lisp, so it's best to be specific.)

Like most *Lisp functions, !! returns a *temporary* pvar, a pvar used to temporarily store the value of a result. Temporary pvars are similar to bits of scrap paper; you don't use them to hold onto important pieces of information, you just use them to temporarily hold onto a value that you're going to store somewhere else.

2.1.2 Permanent Pvars – *defvar

That "somewhere else" is quite often a *permanent* pvar, a pvar defined in such a way that it won't go away until you explicitly deallocate it. The *Lisp operation that allocates permanent pvars is ***defvar**. For example,

```
> (*defvar my-pvar 5)
```

defines a permanent pvar named **my–pvar**, and initializes it to have the value 5 for each processor. (Notice that the scalar value 5 is automatically converted into a pvar. This is true of virtually every operator in *Lisp; scalar values are promoted to pvars where necessary.)

*Lisp automatically defines two permanent pvars for you: **t!!** and **nil!!**. As their names suggest, these are the parallel equivalents of **t** and **nil** in Common Lisp: **t!!** has the value **t** for every processor, and **nil!!** has the value **nil** for every processor.

2.1.3 Local Pvars – *let

As we saw in the preceding chapter, you can also create *local* pvars, that is, pvars that exist only for the duration of a piece of *Lisp code. The *Lisp operator that defines local pvars is ***let**. For example,

```
(*let ((two!! 2)
       (three!! 3.0))
   (+!! two!! three!! my-pvar))
```

defines two local pvars, **two!!** and **three!!**, and takes the parallel sum of these pvars with the permanent pvar **my–pvar** that we defined in the previous section.

2.1.4 Reading, Writing, and Printing Pvar Values

Once you've created a pvar, there are a number of things you can do with it.

- You can examine the value of a pvar for any processor in the machine. The *Lisp operation for this is **pref**, which does a "processor reference." It takes two arguments, a pvar and the number of a particular processor in the CM, and acts much like the Common Lisp operator **aref**, retrieving the value of the supplied pvar for that processor.

As with array elements in Common Lisp, processors are numbered from 0 to one less than the total number of processors you have attached. For example, the form

```
> (pref my-pvar 12)
5
```

returns 5, the value of **my-pvar** in processor number 12. (In fact, since **my-pvar** has the value 5 for every processor, this expression would return 5 no matter which processor you chose to examine.)

- You can change the value of a pvar for any processor. Just as you apply **setf** to **aref** in Common Lisp to change an element of an array, so in *Lisp the operator ***setf** is used in combination with **pref** to change the value of a pvar for a specific processor:

```
> (*setf (pref my-pvar 12) 42)
```

This expression stores the value 42 in **my-pvar** for processor 12. You can check this with a call to **pref**:

```
> (pref my-pvar 12)
42
```

- You can print out the values of a pvar, either for all processors or only for a particular subset. The *Lisp operation to display the values of a pvar is **pretty-print-pvar**, or **ppp** as it is generally known to *Lisp programmers:

```
> (ppp my-pvar :end 20)
5 5 5 5 5 5 5 5 5 5 5 5 42 5 5 5 5 5 5 5
```

This example displays the value of **my-pvar** in the first twenty processors. You can see here the value 42 that we stored in processor 12.

- You can copy values from one pvar to another. The ***set** operator takes two pvar arguments and copies the contents of the second pvar into the first:

```
> (*set my-pvar 9)        ;;; "Set" my-pvar to 9
> (ppp my-pvar :end 20)
9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9

> (*defvar copy-pvar)
> (*set copy-pvar my-pvar) ;;; Copy my-pvar to copy-pvar
> (ppp copy-pvar :end 20)
9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
```

2.2 Data Parallelism–A Different Value for Each Processor

So far, the examples of pvars in this chapter have had the same value in every processor. However, the key point of *Lisp is the *data parallelism* of the language: the ability of each processor to perform the same operation on potentially different data. Most often, the pvars that you create and use in your programs will have a different value for each processor.

*Lisp includes a number of operations that by definition return a different value for each processor. One of these is **random!!**, which we saw in Chapter 1. Another is the function **self-address!!**:

```
> (ppp (self-address!!) :end 20)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

The pvar returned by **self-address!!** contains a special integer for each processor, known as the processor's *send address*. The send address of each processor is unique and constant, so send addresses can be used to refer to individual processors within the machine.

We've already seen one use for send addresses, in the **pref** examples of Section 2.1.4. We'll see further uses later on. For now, just think of **self-address!!** as a handy way to get a pvar that has a different value for every processor in the CM. (I'll use **self-address!!** in future examples to show how the data parallel operations of *Lisp work.)

2.3 Pvar Data Types

*Lisp pvars can contain values of many Common Lisp types. Each of the permissible types is listed below, along with examples of *Lisp expressions that return pvars of that type.

Note: As we've seen, scalar values of any of these data types are automatically promoted to pvars when passed to *Lisp operators that expect pvar arguments.

boolean — Either **t** or **nil** for each processor.

```
t!!        nil!!      (evenp!! (self-address!!))
```

unsigned-byte, signed-byte — An integer (unsigned or signed) for each processor.

```
(+!! 3 5)          (-!! (self-address!!) 800)
```

defined–float — A floating-point number for each processor.

```
(float!! 34)          (/!! (self-address!!) 4)
```

complex — A complex number for each processor.

```
(complex!! 3 1)
```

character — A Common Lisp character for each processor.

```
(!! #\C)              (int-char!! 23)
```

array — A Common Lisp array for each processor.

```
(make-array!! '(2 8) :element-type 'single-float
                 :initial-element pi)
```

structure — A Common Lisp structure object for each processor.

```
(*defstruct particle
     (x-pos 0 :type (unsigned-byte 32))
     (y-pos 0 :type (unsigned-byte 32)))

(make-particle!! 20 61)
```

2.3.1 Other Pvar Types

There are also **front–end** pvars, which contain a reference to a front-end data object for each processor. These are created by the *Lisp **front–end!!** function.

Finally, there is a **general** pvar type that allows you to store values of many different types in the same pvar (with the exception of arrays and structures). Pvars that are not explicitly declared to be of a particular data type are **general** pvars by default. So, for example, the expression

```
(*defvar my-pvar 5)
```

that we used above creates a general pvar. To define a pvar of a specific data type, you must provide a type declaration for the pvar. (We'll see how to do this in Chapter 5.)

2.4 The Size and Shape of Pvars

While you're starting out, you may find it helpful to think of pvars as being a peculiar type of array that just happens to be stored in the memory of the CM. Although there are important differences between Common Lisp arrays and *Lisp pvars, there are a number of similarities as well:

- arrays and pvars both hold many elements
- arrays and pvars have specified sizes and shapes
- the elements of arrays and pvars are accessed by supplying indices
- when passed as arguments to functions, both arrays and pvars are passed by reference, never by value

So if you find yourself having trouble working with pvars, it doesn't hurt to think of them as arrays until you get used to them.

With that said, how can you determine the size and shape of your pvars? The answer is that you determine the size and shape of pvars by setting the size and shape of the current processor grid. There are two ways to do this: by cold-booting the CM with a specific processor grid, and by defining virtual processor sets (VP sets). We'll look at both methods in this section.

2.4.1 Processor Grids and Configurations

When you call ***cold-boot** to attach to a CM and initialize *Lisp, you're not just given a bunch of disorganized processors to play with. The processors of the CM are logically arranged in a one-, two-, or *n*-dimensional grid known as a *processor grid*, or, more generically, a *configuration*.

If you call ***cold-boot** without any arguments, the processors of the CM are arranged in a two-dimensional grid that is as close to being square as the current number of attached processors will allow. For example, for an 8K CM the default grid size is a 64 by 128 grid of processors. If you're using the *Lisp simulator, the default arrangement of processors is an 8 by 4 grid.

This grid-like arrangement of processors will become especially important when we look at processor communication later on, but for right now you can simply think of it as a means of specifying the size and shape of your pvars in the memory of the CM, much as you would specify the size and shape of an array on the front end.

2.4.2 *cold-boot

The ***cold-boot** operator returns two values: the number of attached CM processors and a list of integers that describes the size and shape of the current processor grid.

For example, if you've just attached yourself to an 8K CM, a call to ***cold-boot** will return

```
> (*cold-boot)
8192
(64 128)
```

You can supply arguments to ***cold-boot** to select almost any configuration of processors you want, within limits. In particular, you can select configurations that have more processors than are physically available on the CM to which you've attached. (We'll see some examples of this later on.) Also, ***cold-boot** will "remember" the configuration you specify; if you call ***cold-boot** without any arguments, it will reinitialize *Lisp using the same configuration that it used the last time you called it.

Configuration Variables

Among other things, ***cold-boot** initializes a number of Lisp variables according to the current configuration of processors. Two important examples of these variables are:

- ***number-of-processors-limit***
  The total number of processors in the current configuration.
- ***current-cm-configuration***
  A list of numbers describing the current configuration of the CM processors.

In the 8K ***cold-boot** example shown above, these variables would be initialized as follows:

```
> *number-of-processors-limit*
8192
> *current-cm-configuration*
(64 128)
```

You can use these variables to write *Lisp code that will execute correctly no matter how many CM processors are attached. You can also use these variables to remind yourself of the current "state" of the CM, that is, how many processors you have attached, and what configuration they are in.

2.4.3 Processor Grids and Pvar Shapes

The shape of the current processor grid determines the shape of your pvars. If the value of ***current-cm-configuration*** is **(64 128)**, as in the examples above, then every pvar that you create will have its values arranged in a two-dimensional grid, 64 elements by 128 elements. For example, Figure 5 shows the arrangement of values for the pvar returned by the expression

```
(!! 24)
```

for a 64 by 128 processor grid.

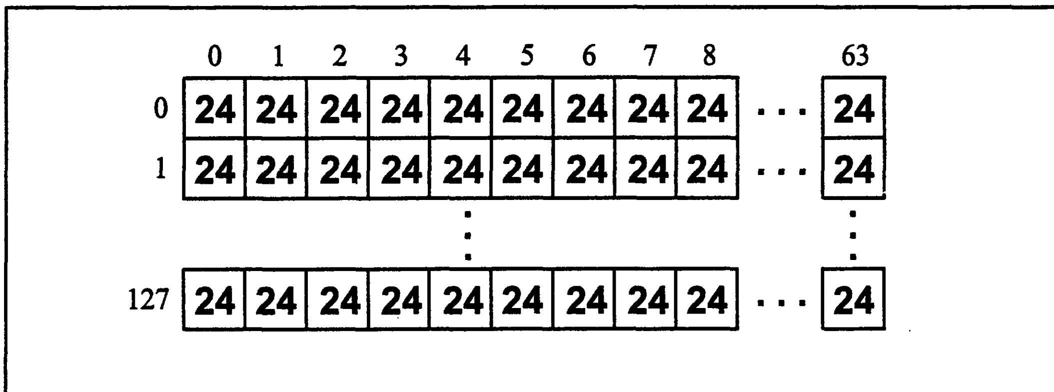


Figure 5. Shape of the pvar returned by **(!! 24)** for a 64 by 128 grid

2.4.4 Pvars of Different Shapes and Sizes—VP Sets

But what do you do if you want to use pvars of different shapes and sizes within the same program? This is where the concept of *virtual processor sets* (VP sets) comes in. You can use VP sets to define differently shaped processor grids that can be used together in the same program. You can then use those configurations to control the shape of your pvars.

We'll see some examples of the creation and use of VP sets later on, but for now let's stick to just using one configuration (the one defined by ***cold-boot**) for all the pvars we define. This will make it easier to focus on the fundamental features of *Lisp.

2.5 Calling Parallel Functions

The *Lisp language includes parallel equivalents for most of the basic operations of Common Lisp. For example, just as there are arithmetic operators in Common Lisp, there are parallel arithmetic operations in *Lisp:

```
> (ppp (+!! (self-address!!) 3) :end 20)
3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22

> (ppp (*!! (self-address!!) pi) :end 6)
0.0 3.1415927 6.2831855 9.424778 12.566371 15.707963

> (ppp (float!! (1+!! (self-address!!))) :end 12)
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0

> (ppp (sin!! (*!! (self-address!!) (/!! pi 4))) :end 6)
0.0 0.7071067 1.0 0.7071074 4.7683707E-7 -0.70710653

> (setq i #C(0.0 1.0))
                                        ;;  2 pi i
> (ppp (exp!! (*!! 2 pi i)) :end 5)     ;; e        = 1
#C(1.0 0.0) #C(1.0 0.0) #C(1.0 0.0) #C(1.0 0.0) #C(1.0 0.0)

> (ppp (random!! 20) :end 20)
11 11 18 6 19 10 5 3 4 1 3 10 3 6 5 9 7 5 6 19
```

In general, the *Lisp parallel equivalent of a Common Lisp operator

- has the same name, with either "!!" added to the end, or "*" added in front
- performs the same, or nearly the same operation, but in parallel on the CM

There are far more parallel operators of this variety than we have space to examine here. If you want to know whether a particular Common Lisp function has a *Lisp equivalent, see the **Lisp Dictionary*, which includes a complete list of the functions and macros available in *Lisp.

For the Curious: What do the !!'s and *'s mean? As a general rule of thumb, !! (pronounced "bang-bang") is used to mark *Lisp functions that always return a pvar, while * (pronounced "star") is used to mark parallel operators that may or may not return a pvar. With one or two minor exceptions, this rule is followed consistently throughout the *Lisp language. The name of the language itself, "star-Lisp," comes from this convention.

2.6 Printing Pvars in Many Ways

Let's stop here for a moment and take a closer look at the pvar printing function **ppp**. The **ppp** operator has a large number of keyword arguments that let you specify exactly how you would like the values of a pvar to be printed out.

The simplest method is the way we've been doing it so far, that is, using the **:end** keyword to say where **ppp** should stop displaying values:

```
> (ppp my-pvar :end 20)
5 5 5 5 5 5 5 5 5 5 5 5 42 5 5 5 5 5 5 5
```

You can also specify a **:start** point, if you'd like to take a look at some values in the middle of a pvar:

```
> (ppp (self-address!!) :start 20 :end 30)
20 21 22 23 24 25 26 27 28 29
```

You can display a fixed number of pvar values **:per-line**,

```
> (ppp (self-address!!) :end 30 :per-line 12)
0 1 2 3 4 5 6 7 8 9 10 11
12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29
```

or, if you want to see how the values of a pvar are arranged on the current processor grid, **ppp** has a **:mode** argument that lets you ask for the **:grid** view of things:

```
> (ppp (self-address!!) :mode :grid :end '(4 4))

DIMENSION 0 (X)  ----->

0 2 4 6
1 3 5 7
8 10 12 14
9 11 13 15
```

If the display looks a little disorderly, as in the example above, you can use the **:format** argument to control the format in which values are printed:

```
> (ppp (self-address!!) :mode :grid :end '(4 4) :format "~2D ")

 DIMENSION 0 (X)  ----->

 0  2  4  6
 1  3  5  7
 8 10 12 14
 9 11 13 15
```

One more useful trick: If you want to, you can add a **:title** to your output:

```
> (ppp (self-address!!) :end 32 :per-line 8
      :format "~2A " :title "Send addresses")

Send addresses:
0  1  2  3  4  5  6  7
8  9  10 11 12 13 14 15
16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31
```

2.7 Defining *Lisp Functions and Macros

Now that we've seen some of the parallel functions available in *Lisp, and seen how to print out the pvars they return, let's look at how you can define your own parallel functions and macros.

2.7.1 To Define a Function, Use defun

If you've written Common Lisp programs before, then the operator used to define parallel functions in *Lisp should look very familiar to you. It's called **defun**:

```
(defun add-mult!! (a b c)
   (*!! (+!! a b) c))
```

This example defines a function called **add-mult!!** that takes three arguments, either pvars or numeric values. In each processor on the CM, **add-mult!!** adds the values of the first two arguments (**a** and **b**) and then multiplies by the value of the third argument (**c**).

*Lisp functions defined by **defun** are called exactly as you would call any other Lisp function. So, for example,

```
> (ppp my-pvar :end 12)
9 9 9 9 9 9 9 9 9 9 9 9

> (*set my-pvar (add-mult!! my-pvar -6 (self-address!!)))
NIL
```

This call to **add-mult!!** subtracts 6 from the value of **my-pvar** for each processor, then multiplies by the value of **(self-address!!)**. The result is stored back into **my-pvar** by ***set**, as we can see by calling **ppp**:

```
> (ppp my-pvar :end 12)
0 3 6 9 12 15 18 21 24 27 30 33
```

*Lisp functions defined by **defun** are no different from any other Lisp function. You can use **apply** and **funcall** to make calls to them, and use the Common Lisp tracing functions **trace** and **untrace** to track calls to them.

For the Observant: You may have noticed that *Lisp includes a special-purpose variant of **defun** called ***defun**. However, for most purposes ***defun** isn't necessary; **defun** should be used instead. For more information on the difference between these operators, see the entry on ***defun** in the **Lisp Dictionary*.

2.7.2 To Define a Macro, Use defmacro

The operation used to define macros in *Lisp should also look familiar to Common Lisp programmers. It's called **defmacro**:

```
> (defmacro *two-incf (pvar)
      `(*set ,pvar (+!! ,pvar 2)))
*TWO-INCF
```

This macro takes a single pvar argument and increments its value by 2 for each processor.

*Lisp macros defined by **defmacro** are called exactly as you would call any other Lisp macro. For example:

```
> (*two-incf my-pvar)
NIL
> (ppp my-pvar :end 12)
2 5 8 11 14 17 20 23 26 29 32 35
```

2.8 Compiling *Lisp Code

*Lisp functions and macros are compiled exactly as they are in Common Lisp. This means that there are three basic ways to compile a *Lisp function:

- You can call the Common Lisp function **compile** to compile a specific function:

  ```
  (compile 'add-mult!!)
  ```

- You can call the Common Lisp function **compile-file** to compile a file of code, which you can then **load** into *Lisp:

  ```
  (compile-file "starlisp-code.lisp")
  (load "starlisp-code")
  ```

- You may also be able to use your editor to compile your code. Depending on the Lisp programming tools your editor provides, there may be a special keystroke you can use to ask the editor to compile a function definition. In Emacs-style editors this keystroke is commonly Ctrl-Shift-C or Meta-Shift-C. Check the manual for your editor for more information.

As in Common Lisp, compilation of code is optional. You don't have to compile your code before you can run it. Use the compilation method that's most comfortable for you, or don't compile at all, if you choose not to.

A Note about Declarations

One important difference between Common Lisp and *Lisp is that *Lisp requires complete, explicit declarations of the data types of variables and the returned values of functions in order to fully compile *Lisp code.

Declarations in *Lisp look much like the standard Common Lisp declarations, except that there are additional data type specifications for pvars. These additional type specifiers are described in Chapter 5.

For example, a completely declared version of **add-mult!!** might look like

```
(defun add-mult!! (a b c)
   (declare (type (pvar (signed-byte 32)) a b c))
   (*!! (+!! a b) c))
```

This doesn't mean that if you don't provide declarations, your code won't compile. You just won't see the full potential of *Lisp for speed and efficiency in your compiled code.

For the most part you can ignore type declarations while you're learning *Lisp. We'll take a deeper look at both declarations and the *Lisp compiler in Chapter 5.

2.9 Summary: How *Lisp and Common Lisp Are Similar

In this chapter, we've seen most of the features of *Lisp that are similar to things you'll already have encountered in Common Lisp:

- parallel variables (pvars), which are the *Lisp versions of variables in Common Lisp, and resemble arrays in their use
- the pvar printing operator **ppp**, which is used to display the contents of pvars
- parallel functions that cause each CM processor to perform a Common Lisp operation
- the operators **defun** and **defmacro**, which are used in *Lisp just as they are in Common Lisp, to define functions and macros
- the *Lisp compiler, an extension of the Lisp compiler for compiling *Lisp code

In the next chapter, we'll take a look at the parallel programming tools of *Lisp that are the real heart of the language.

"There is always work,
And tools to work withal, for those who will..."
James Russell Lowell

Chapter 3

Parallel Programming Tools

Up until this point, we've been using the CM simply as a massively parallel calculator, having every processor perform the same operation at the same time. From here on we'll see how we can use the CM as a massively parallel computing system, by using the parallel programming tools of *Lisp.

In Chapter 2, we saw the features of *Lisp that are similar to Common Lisp. Now we'll look at the CM-specific parallel programming tools of *Lisp that distinguish it from Common Lisp.

3.1 Data Parallel Programming

The *Lisp tools for data parallel programming fall into five main categories:

- tools for selecting a set of processors to perform an operation
- tools for moving data between processors within the CM
- tools for moving data between the CM and the front end
- tools for combining and transforming data
- tools for determining the shape of your pvars (VP sets)

In the following sections, we'll look at one or two examples from each category.

3.2 Processor Selection Operators

Because you don't always want to perform the same operation in every processor, *Lisp allows you to select which processors will perform a given operation. Each processor maintains an internal flag that determines whether it is *active*, that is, whether or not it will execute the instructions it receives. Because of this, you can choose a subset of the processors in the machine, known as the *currently selected set*, to execute any given operation. The rest of the processors are *deselected*, and do nothing.

The processor selection operators in *Lisp are modeled after the conditional tests of Common Lisp, and typically select processors based on the result of a parallel test. By default, all CM processors are active when you ***cold–boot** *Lisp, so the main job of the processor selection operators is to temporarily turn off, or *deselect*, some processors for the duration of a piece of code.

3.2.1 Processor Selection — Doing More by Doing Less

"What's the good of that?" you might ask. "The point of having a massively parallel CM is to have all the processors computing at once. If I turn some of them off I'm wasting computing power, right?" In a sense, yes; if you were to turn off all but one of the CM processors, that certainly wouldn't be very efficient. However, it's unlikely that you would ever knowingly choose to do that.

A better way to look at it is the way Michelangelo looked at carving a statue of a lion: He selected all those parts of a block of stone that weren't lion, and carved them away. Processor selection operators let you select just those values of a pvar that have a certain important property, so that you can perform an operation on those values while leaving the rest undisturbed. What you temporarily lose in computing power, you more than gain back in computing finesse.

For example, a *Lisp function to carve Michelangelo's lion would look like:

```
(defun carve-lion (stone-pvar)
   (*when (not!! (lion-p!! stone-pvar))
      (*set stone-pvar (carve!! stone-pvar))))
```

The *Lisp operation ***when** uses a parallel test to determine which processors should be active. In this case, the test **(not!! (lion–p!! stone–pvar))** selects only those processors where **stone–pvar** doesn't look like a lion. The ***set** form is evaluated only for those processors, with the result that all values of **stone–pvar** that are **not!! lion–p!!** are **carve!!**-ed away, leaving us a **stone–pvar** with a single perfect lion inside.

This function would actually work, if we could find someone like Michelangelo to write the **lion-p!!** test for us. Failing that, let's look at a more mundane example, in which we'll use ***when** to perform a calculation using only the even-numbered CM processors:

```
> (*defvar data 4)
DATA
> (ppp data :end 20)
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
```

Here, the pvar **data** is given the value 4 in every processor. We'll need to use the value of the **self-address!!** function, so let's print out its value for a few processors:

```
> (ppp (self-address!!) :end 20)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

Let's assume we want to add 3 to **data** in every even-numbered processor. That means that we need a test that returns **t** wherever the value of **self-address!!** is even. *Lisp has such a test: **evenp!!**

```
> (ppp (evenp!! (self-address!!)) :end 20)
T NIL T NIL T NIL T NIL T NIL T NIL T NIL T NIL T NIL T NIL
```

With this test in hand, the actual ***when** expression is simple:

```
> (*when (evenp!! (self-address!!))
      (*set data (+!! data 3)))
NIL
> (ppp data :end 20)
7 4 7 4 7 4 7 4 7 4 7 4 7 4 7 4 7 4 7 4
```

The result, as we can see, is that the value of **data** is set to 7 for every even-numbered processor. Not much like a lion, but it's a start.

A related parallel conditional, **if!!**, also selects processors based on a test. The difference is that **if!!** also returns a pvar result:

```
> (ppp (if!! (oddp!! (self-address!!))
             (+!! data 3)
             (-!! data 3))
     :end 20)
4 7 4 7 4 7 4 7 4 7 4 7 4 7 4 7 4 7 4 7
```

As you can see in this example, **if!!** actually performs two selections:

- First, all odd processors are selected (that is, all processors in which **(oddp!! (self-address!!))** is **t**), and the *sum* of **data** and 3 is calculated for those processors.
- Then those processors are deselected, and all the remaining processors (those for which **(oddp!! (self-address!!))** is **nil**) are selected. In these processors, the *difference* of **data** and 3 is calculated.

The value returned by **if!!** for each processor is the value it obtains from one or the other of these two operations. In this example, **if!!** returns the sum of **data** and 3 in all odd-numbered processors, and the difference of **data** and 3 in all even-numbered processors. The 7's and 4's in the pvar returned by **if!!** thus alternate in a manner exactly opposite that of the original **data** pvar.

For the Curious: Enumerating Selected Processors

One *Lisp function that comes in handy for working with selected sets of processors is **enumerate!!**. This function is similar to **self-address!!**, in that it assigns unique sequential integer values to processors. The difference is that **enumerate!!** assigns values only to selected processors:

```
> (*defvar empty-pvar 0)
EMPTY-PVAR

> (ppp empty-pvar :end 20)
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

> (*when (evenp!! (self-address!!))
      (*set empty-pvar (enumerate!!)))
NIL
> (ppp empty-pvar :end 20)
0 0 1 0 2 0 3 0 4 0 5 0 6 0 7 0 8 0 9 0
```

3.2.2 The Processor Selection Operators of *Lisp

There are *Lisp processor selection operators that correspond to all the major Common Lisp conditionals, including **when**, **unless**, **if**, **cond**, **case**, and **ecase**, plus two additional CM-specific operations, ***all** and **with-css-saved**. We don't have room to examine each of them here, but they are described in more detail in the **Lisp Dictionary*.

3.3 Communication Operators

When applied to processors on the CM, the term "communication" essentially means "moving data around between processors." There are two main methods of communication in *Lisp: *router* communication and *grid* communication.

3.3.1 Router Communication — General-Purpose Data Exchange

The processors in the CM are all linked together by a general message-passing network called the *router*. The router allows each processor to send a piece of data to any other processor in the CM, with all processors transmitting data simultaneously.

A useful way to picture this is to think of the router as a postal service for processors (see Figure 6).

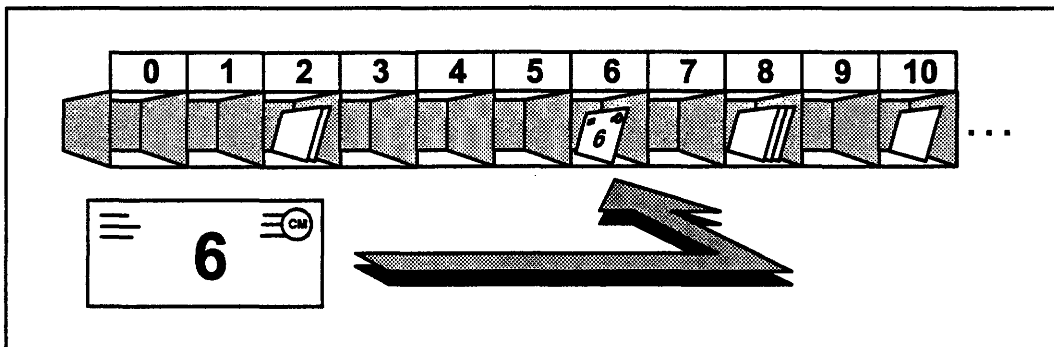


Figure 6. The router handles messages much like a postal service

Each processor has a unique router address, which we've already seen; it's the value returned by **(self-address!!)**. When one processor needs to send a piece of data to another processor, the sending processor needs to supply only two pieces of information: the data itself, and the address of the processor that should receive it. The router takes it from there, simultaneously accepting messages from every processor in the CM and making sure that all messages are delivered to their destinations as quickly as possible.

To carry the post-office metaphor a step further, every router exchange involves three pvars: one that holds the message being sent by each processor, another that holds the address to which each message should be delivered, and a third pvar that serves as the set of "mailboxes" into which the messages are deposited.

3.3.2 The Router Communication Operators of *Lisp

The router communication operators in *Lisp are:

- ***pset**, which does a parallel "send", in which each processor sends a value to another processor
- **pref!!**, which does a parallel "get", in which each processor requests a value from another processor

For example, a call to ***pset** looks like:

```
> *number-of-processors-limit*
8192
> (*pset :no-collisions (self-address!!)
                        data
                        (-!! *number-of-processors-limit*
                             (self-address!!)
                             1))
> (ppp data :end 10)
8191 8190 8189 8188 8187 8186 8185 8184 8183 8182
```

Note: This example shows output for an 8K CM. You may see different results, depending on the size of your CM. If you're using the *Lisp simulator, the numbers you see displayed will be much smaller. Nevertheless, the first value printed by **ppp** will be one less than ***number-of-processors-limit***, with the following values in descending order.

In this example:

- A call to **(self-address!!)** is used as the pvar of messages to be sent;. that is, each processor is sending its own address.
- The **data** pvar is the set of mailboxes into which the messages are delivered.
- The expression **(-!! *number-of-processors-limit* (self-address!!) 1)** provides the addresses; that is, processors with low send addresses send to processors with high send addresses, and vice versa.

The result is that the **data** pvar now contains a copy of the **(self-address!!)** pvar with its values "flipped" end for end.

For the Curious: What's the **:no-collisions** keyword there for? Since no two messages are being sent to the same processor, we can use the **:no-collisions** keyword argument of ***pset** to let it know that it can use the most efficient communication algorithms possible.

3.3.3 Grid Communication — Fast, but with Restrictions

There's one main problem with router communication; just like the real-life Postal Service, for some kinds of communication it's just too slow. For this reason, the CM includes a more specialized form of communication called *grid communication.*

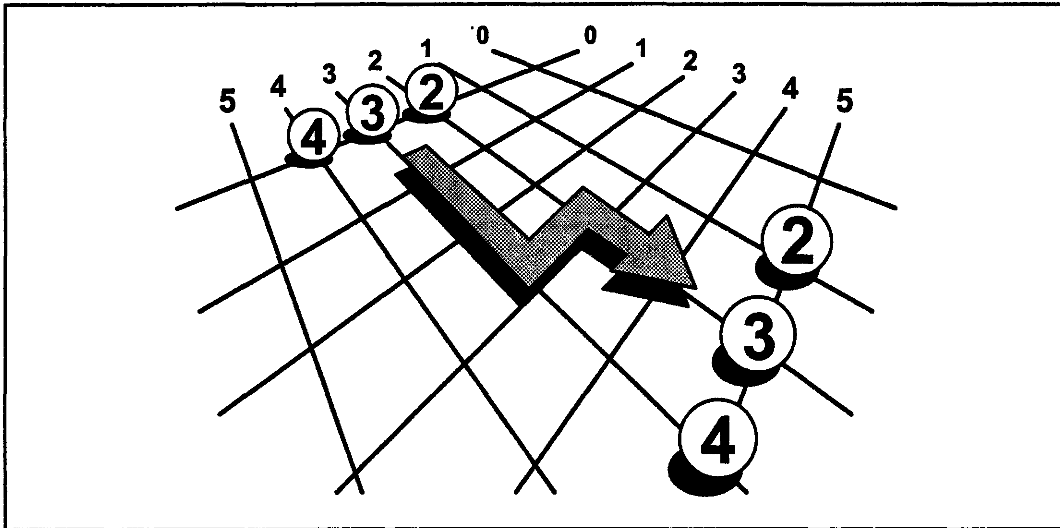


Figure 7. In grid communication data moves along the axes of the processor grid

Whereas router communication allows each processor to send a message to any other processor in the machine, in grid communication all processors send their messages in the same direction along the axes of the current processor grid (see Figure 7).

Because the messages all "follow the lines" of the processor grid, so to speak, and because the movement of every message is expressed in terms of "so many processors up," "so many processors over," and so on, grid communication is generally much faster than router communication.

Grid communication does limit you to moving your data across the processor grid in lock-step fashion, much like soldiers marching in formation on a parade ground, but for many applications, such as the cellular automata example we saw in Chapter 1, this grid-like motion of data is exactly what is needed.

3.3.4 The Grid Communication Operators of *Lisp

The main grid communication operators of *Lisp are:

- **news!!**, which shifts values across the grid according to a set of relative coordinates
- ***news**, which shifts values, and also stores them in a supplied destination pvar

As an example, let's look at ***news**. We can use ***news** to write a function that performs the zig-zag movement of data shown in Figure 7 as follows:

```
> (defun zig-zag (pvar)
      (*news pvar pvar 3 0)    ;;; three steps "east"
      (*news pvar pvar 0 -1)   ;;; one step "north"
      (*news pvar pvar 2 0))   ;;; two steps "east"
ZIG-ZAG
```

Each of the ***news** calls in this function takes data from the supplied **pvar**, shifts it a number of steps across the grid, and then stores it back in **pvar**. The two numeric arguments in each ***news** call specify how many steps to take along each grid axis.

Of course, you're not limited to motion along a single axis. In the example above I've divided up the movement of data into three steps to emphasize the stepwise nature of grid communication. You can just as easily define the **zig-zag** function with a single ***news** call:

```
> (defun zig-zag (pvar)
     (*news pvar pvar 5 -1))  ;;; five steps "east", one "north"
ZIG-ZAG
```

For the Curious: You might wonder why it's called "***news**," and not something more memory-jogging like "***grid**." The reason is historical. Grid communication is often referred to as "NEWS" ("North-East-West-South") communication, because on early versions of the CM hardware, grid communication was limited to two-dimensional processor grids. The grid communication operators of *Lisp likewise derive their names from this two-dimensional pattern of data exchange.

3.3.5 An Example of Grid Communication

Let's define a pvar and use the **zig–zag** function to move its values around. We'll also use the **:grid** argument to **ppp** so that we can see the current processor grid displayed as a grid of values on the screen.

For Simulator Users: The following examples assume that the current processor grid is two-dimensional and at least 6 by 6 processors in size. If you're running *Lisp on CM hardware this is the default. If you're using the *Lisp simulator, however, the default processor grid (8 by 4) won't be large enough, so type the following:

```
> (*cold-boot :initial-dimensions '(8 8))
```

Let's create a permanent pvar with the value 0 in every processor, and then store three integers into it so we can see the movement of data between processors:

```
> (*defvar data-pvar 0)
DATA-PVAR
> (*setf (pref data-pvar (grid 0 2)) 2)
> (*setf (pref data-pvar (grid 0 3)) 3)
> (*setf (pref data-pvar (grid 0 4)) 4)
```

Here again, we're using the function **grid** to refer to processors by their grid coordinates. Now let's use **ppp** to display the **data–pvar** in **:grid** format:

```
> (ppp data-pvar :mode :grid :end '(6 6))
0 0 0 0 0 0
0 0 0 0 0 0
2 0 0 0 0 0
3 0 0 0 0 0
4 0 0 0 0 0
0 0 0 0 0 0
```

Here we're displaying the top left-hand "corner" of the current processor grid, and we can see the values 2, 3, and 4 that we stored into **data–pvar** in the left-most column.

Now in a dazzling display of computational power, we'll use **zig–zag** to move them:

```
> (zig-zag data-pvar)
NIL

> (ppp data-pvar :mode :grid :end '(6 6))
0 0 0 0 0 0
0 0 0 0 0 2
0 0 0 0 0 3
0 0 0 0 0 4
0 0 0 0 0 0
0 0 0 0 0 0
```

Of course, you'll typically use grid communication operators to move many more data values around the grid than the three integers we've used here. The principles, however, remain the same.

3.4 Front-End/CM Communication

*Lisp includes three types of operators that let you move data quickly between the front end and the CM:

- The simplest type is the operator **!!**, which as we've seen takes a single front-end value and "broadcasts" it to every processor in the CM.
- *Lisp also has a set of "global" communication operators that do just the opposite: they take a pvar on the CM and turn it into a single front-end value.
- Finally, *Lisp has a number of array/pvar conversion functions that make it easy for you to move large amounts of data between the front end and the CM.

In this section, we'll take a look at examples of the latter two types of operators.

3.4.1 Funnels for Data—Global Communication Functions

The global communication functions all have one feature in common: They take a pvar argument, combine the values of that pvar into one value, and then return that value to the front end. You can think of these operators as computational funnels; they take a large set of values from the CM and combine them into a single front-end value.

A good example is ***sum**, which adds together all the values of a pvar:

```
> (*defvar numbers (random!! 10))
NUMBERS
> (ppp numbers :end 20)
2 1 8 7 4 5 8 2 1 8 3 7 3 0 9 6 4 9 4 9

> (*sum numbers)
18651
```

One handy use of ***sum** is based on the fact that global functions only combine values from active processors (processors currently selected by an operator such as ***when** or **if!!**). If you want to know how many processors are currently active, a simple way to find out is to take the ***sum** of 1:

```
> (*sum 1)   ;;; on an 8K machine (8192 processors)
8192
> (*when (evenp!! (self-address!!)) ;;; turn off odd processors
      (*sum 1))
4096
```

Other useful global operations are:

- ***max, *min** — find the maximum and minimum values of a pvar

  ```
  > (*max (self-address!!)) ;; Returns highest send address.
  8191
  ```

- ***and, *or** — find the logical AND/inclusive OR of the values of a pvar

  ```
  > (*or t!!) ;; Returns T if any processors are selected
  T
  > (*when nil!! (*or t!!)) ;; and NIL if none are selected
  NIL
  ```

3.4.2 Bulk Data Movement—Array/Pvar Conversions

If you want to move a large amount of data from the front end to the CM or vice versa, you can call the array/pvar conversion functions of *Lisp, which are:

- **array–to–pvar** — converts an array on the front end to a pvar on the CM
- **pvar–to–array** — converts a pvar on the CM to an array on the front end

For example, given the array and pvar defined by

```
> (defvar data-array #(1 2 3 4 5 6))     ;;; Front-end array

> (*defvar data-pvar 0)                  ;;; Empty pvar on the CM
```

we can use **array–to–pvar** to copy the six elements of **data–array** into the first six values of **data–pvar**:

```
> (array-to-pvar  data-array data-pvar :end 6)

> (ppp data-pvar :end 12)
1 2 3 4 5 6 0 0 0 0 0 0
```

Here the **:end** keyword argument is used (as with **ppp**) to specify the send address at which the copying of values ends.

We can use **pvar–to–array** to copy the first three values of **data–pvar** back into **data–array**, starting at element 3:

```
> (pvar-to-array  data-pvar data-array :array-offset 3
                  :start 0 :end 3)

> (let ((*print-array* t))
    (print data-array))
#(1 2 3 1 2 3)
```

Here we're using both the **:start** and **:end** keywords to specify a range of pvar values to be copied. The **:array–offset** keyword is used to specify element 3 of the array as the first element into which a value is copied.

For more examples of these functions, see the entries for them in the **Lisp Dictionary*.

3.5 Parallel Data Transformation in *Lisp

*Lisp includes operators that allow you to perform large-scale transformations of your data. For example, you can use *Lisp operations to take a cumulative sum of the values of a pvar across the current processor grid. In this section, we'll look at the *Lisp operators that allow you to perform these kinds of parallel data manipulations.

The data transformation functions of *Lisp include tools for:

- performing cumulative operations (such as the cumulative sum mentioned above)
- copying important pvar values across the processor grid
- ranking and sorting the values of a pvar

3.5.1 Parallel Prefixing (Scanning)

Scanning is a transformation in which a cumulative operation is performed on the values of a pvar across the currently selected grid. The main scanning function of *Lisp is **scan!!**. The **scan!!** function has an extensive set of options that let you choose from a number of possible scanning operations, such as addition or multiplication of values; taking the maximum and minimum of values; taking the logical or arithmetic AND, OR, and XOR of values; and even simply copying values across the processor grid.

A common use of the **scan!!** operation is for cumulative summations. For example:

```
> (ppp (scan!! 2 '+!!) :end 20)
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40
```

Here, each processor calculates the sum of the values for all processors up to and including itself, producing a cumulative sum of the values of the pvar.

The **scan!!** operator has a keyword argument, **:include-self**, that lets you specify whether each processor will include its own value in the calculation. For example, here's the above addition with the **:include-self** argument of **nil**:

```
> (ppp (scan!! 2 '+!! :include-self nil) :end 20)
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38
```

In this case, each processor calculates the sum of the pvar values for all processors preceding but not including itself. As shown in these examples, you can use the **:include-self** argument to "shift" the result of a scan over by one processor.

A unique feature of **scan!!** is that it allows you to select segments of a pvar over which independent scans are performed. The **scan!!** operation uses a **:segment-pvar** argument to select pvar segments.

An example of **scan!!** with a **:segment-pvar** argument is:

```
> (*defvar segments (zerop!! (mod!! (self-address!!) 4)))

> (ppp segments :end 16)
T NIL NIL NIL T NIL NIL NIL T NIL NIL NIL T NIL NIL NIL

> (ppp (scan!! 1 '+!! :segment-pvar segments) :end 16)
1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4
```

Every **t** value in the **:segment-pvar** argument of **scan!!** indicates the start of another segment, and as you can see the summation is done only within segments.

The **scan!!** function is a powerful programming tool, and has some surprising uses. We'll see some more examples of **scan!!** in the sample *Lisp functions of Chapter 6.

3.5.2 Spreading Values across the Grid

An operation related to scanning is spreading. Spreading copies the values from one "row" or "column" of the current grid to all rows or columns of the grid. (Of course, when you're using processor grids with more than two dimensions, the terms "row" and "column" don't really apply, but the principle is the same.)

The *Lisp operator for spreading is called **spread!!**. As an example, let's assume that **(self-address!!)** currently returns the values shown here:

```
>   (ppp   (self-address!!)
           :mode :grid
           :end '(8 3)
           :format "~3S ")
0    1    2    3    16   17   18   19
4    5    6    7    20   21   22   23
8    9    10   11   24   25   26   27
```

and let's say that we want to spread the values in the fourth column to all of the columns. (What this means in terms of the processor grid is that we want to spread values from the processors at coordinate 3 of dimension 0 to all processors along dimension 0.)

The following call to **spread!!** will do this for us:

```
> (ppp (spread!! (self-address!!) 0 3) ;; Dimension 0, Coord. 3
          :mode :grid
          :end '(9 3)
          :format "~3S ")
3   3   3   3   3   3   3   3   3
7   7   7   7   7   7   7   7   7
11  11  11  11  11  11  11  11  11
```

The **spread!!** function is most useful in combination with communication and scanning operators, because it allows you to quickly spread important values across the processor grid.

3.5.3 Sorting Pvar Values

One other useful transformation operator is **sort!!**, which sorts the values of a numeric pvar in ascending order. For example:

```
> (*defvar random-numbers (random!! 10))
RANDOM-NUMBERS

> (ppp random-numbers :end 20)
3 3 5 8 2 8 9 3 5 5 4 4 1 2 7 9 8 6 9 7

> (*when (<!! (self-address!!) 20)
   (ppp (sort!! random-numbers '<=!!) :end 20))
1 2 2 3 3 3 4 4 5 5 5 6 7 7 8 8 8 9 9 9
NIL
```

Here I've used ***when** to select the first 20 values of the **random-numbers** pvar, and then called **sort!!** to return a copy of those values sorted in ascending order.

If you prefer to write your own sorting routines, *Lisp also includes an operator called **rank!!** that simply returns a numerical ranking of the values of a pvar. For example:

```
> (*when (<!! (self-address!!) 20)
   (ppp (rank!! random-numbers '<=!!) :end 20))
3 4 8 14 1 15 17 5 9 10 6 7 0 2 12 18 16 11 19 13
```

3.6 Configuration Operators

We've seen that the shape of the current processor grid determines the size and shape of your pvars. *Lisp includes several functions and macros that let you control the size and shape of the current processor grid, as well as tools that let you define multiple grids that can be used together in the same program. You've already seen one of these operators: ***cold–boot**. In this section we'll take a look at these operators, and see how you can use them in your programs to control the current configuration of the CM.

3.6.1 Defining the Initial Grid Configuration — *cold–boot

Whenever you initialize *Lisp with the function ***cold–boot**, you're also choosing the size and shape of the current processor grid. The ***cold–boot** function has a keyword argument, **:initial–dimensions**, that determines the initial shape and size of the processor grid.

For example, if you wanted to start off with a cube-shaped grid of processors, 64 processors on a side, you could type the following:

```
> (*cold-boot :initial-dimensions '(64 64 64))
8192
(64 64 64)
```

This gives you a rather mind-boggling number of processors to work with—more than a quarter million, as a matter of fact.

```
> *current-cm-configuration*
(64 64 64)
> *number-of-processors-limit*
262144
```

For Simulator Users: I don't recommend trying this example on the *Lisp simulator! The number of processors that the simulator can handle is limited only by the speed and memory capacity of your computer, but this means that the more processors you use, the slower your program will run. A better example for simulator users is:

```
> (*cold-boot :initial-dimensions '(8 8 8))
Thinking Machines Starlisp Simulator.  Version 18.0
1
(64 64)
```

3.6.2 Where Did the Extra Processors Come From?

Notice that even though I've requested a large number of processors in these examples, many more than the number of physical processors available in the machine, the actual number of physical processors hasn't changed—the first argument returned by ***cold-boot** is still 8192. (As in previous ***cold-boot** examples, I'm assuming an 8K CM.) By requesting a larger number of processors than are present in the physical hardware of the CM, you automatically tell the CM to use *virtual processors*.

Whenever you request a number of processors that is larger than the actual number of physical processors within the CM, each physical processor simulates the operations of two or more *virtual processors* (or VPs, as they are commonly called).

This simulation takes place automatically and transparently. The only effects you're likely to notice are a change in execution speed and a reduction in the total number of pvars you can create. (When you use virtual processors, more than one pvar value is stored in the memory of each physical processor, so your pvars take up more memory.)

3.6.3 Defining Custom Configurations — VP Sets

Once you start working with data sets of different sizes, you'll want to make use of this ability to define processor grids of various shapes and sizes, because the shape and size of the current processor grid determines the shape and size of your pvars.

*Lisp allows you to define special data objects called *virtual processor sets* (VP sets) that describe particular configurations of processors. You can then use *Lisp operations to select these configurations when you need them.

For example,

```
> (def-vp-set big-square '(256 256))
```

defines a two-dimensional VP set called **big-square**, which describes a two-dimensional grid of processors of size 256 by 256. Once you've created a VP set such as this, you can start using it to define pvars with the same size and shape:

```
> (*defvar big-square-pvar 1 nil big-square)
BIG-SQUARE-PVAR
> (*defvar another-square-pvar (random!! 20) nil big-square)
ANOTHER-SQUARE-PVAR
```

You can even save a step and define both the VP set and the pvars at the same time. For example, the three examples above could be replaced by the single form:

```
(def-vp-set big-square '(256 256)
     :*defvars '((big-square-pvar 1)
                 (another-square-pvar (random!! 20))))
```

For the Curious: Why are they called VP "sets"? Because VP sets are used to determine the size and shape of pvars. For each VP set that you define there is an associated "set" of pvars that have the same size and shape. It's the combination of a particular arrangement of virtual processors and a group of pvars whose shape is specified by that arrangement that we mean by the term "VP set". The **def-vp-set** operator makes this relationship explicit, by allowing you to define both a VP set and its associated pvars at the same time.

3.6.4 Default and Current VP Sets

How do you use these VP sets, once you've defined them? In a sense, you've already been using them, ever since you started up *Lisp. When you ***cold-boot** *Lisp, a special *default VP set* is defined that represents whatever processor grid you've chosen to start with (that is, whatever grid size and shape you've specified via the **:initial-dimensions** argument).

If you always used the default VP set and never created VP sets of your own, you probably wouldn't even notice that VP sets existed at all. However, once you've defined your own VP sets, you can use them to switch between different processor configurations within a single *Lisp program.

At any time, there is always one VP set that is the *current VP set*, the one that determines how the processors of the CM are currently arranged. You make a particular VP set the current one by *selecting* it. Unless you specify otherwise, all pvars are created with the shape and size of the current VP set.

The default VP set is selected automatically for you by ***cold-boot**. You can then use the VP set operations of *Lisp to temporarily or permanently select a VP set of your own.

3.6.5 Selecting VP sets

When you want to use a particular VP set, such as the **big-square** defined above, you can either select it temporarily (for the duration of a piece of code), via the macro ***with-vp-set**:

```
> (*cold-boot :initial-dimensions '(128 128))
8192
(128 128)

;;; Display the shape and size of the current processor grid
> (list  *current-cm-configuration*
         *number-of-processors-limit*)
((128 128) 16384)

;;; Display shape and size of the grid with big-square selected
> (*with-vp-set big-square
     (list  *current-cm-configuration*
            *number-of-processors-limit*))
((256 256) 65536)

;;; When *with-vp-set exits, the old grid is restored
> (list  *current-cm-configuration*
         *number-of-processors-limit*)
((128 128) 16384)
```

or you can select it permanently (until you select another VP set), by using the **set-vp-set** function:

```
> (set-vp-set big-square)
#<VP-SET Name: BIG-SQUARE, Dimensions (256 256) ... >

;;; Display shape and size of VP set selected by set-vp-set
> (list *current-cm-configuration*
         *number-of-processors-limit*)
((256 256) 65536)
```

3.6.6 Two Important VP Set Variables

There will often be times that you'll want to know exactly which VP set is the current one, as well as which VP set is the default defined by ***cold-boot**. For this reason, *Lisp maintains two important global variables that keep track of these details.

The *Lisp variable ***default-vp-set*** is always bound to the default VP set defined for you by ***cold-boot**:

```
> *default-vp-set*
#<VP-SET Name: *DEFAULT-VP-SET*, Dimensions (128 128) ... >
```

This variable is automatically set each time you call ***cold-boot**, but otherwise its value doesn't change.

The variable ***current-vp-set*** is always bound to the currently selected VP set, and its value changes every time the current VP set changes:

```
> *current-vp-set*
#<VP-SET Name: BIG-SQUARE, Dimensions (256 256) ... >

> (set-vp-set *default-vp-set*)
#<VP-SET Name: *DEFAULT-VP-SET*, Dimensions (128 128) ... >

> *current-vp-set*
#<VP-SET Name: *DEFAULT-VP-SET*, Dimensions (128 128) ... >
```

You can compare these variables to test whether the current VP set is the default:

```
> (defun default-p ()
    (eq *current-vp-set* *default-vp-set*))

> (set-vp-set big-square)
#<VP-SET Name: BIG-SQUARE, Dimensions (256 256) ... >
> (default-p)
NIL
> (set-vp-set *default-vp-set*)
#<VP-SET Name: *DEFAULT-VP-SET*, Dimensions (128 128) ... >
> (default-p)
T
```

VP sets are an advanced *Lisp tool, and there are many more uses for them than we have room to describe here For an example of the use of VP sets, see Section 6.4 in Chapter 6.

3.7 Summary: *Lisp as a Language

Here's a quick review of what we've covered in this chapter:

- Processor selection operators, such as ***when** and **if!!**, let you increase the power of your *Lisp programs by restricting the set of processors that are currently active.
- Communications operators, such as ***pset** and ***news**, let you move data between processors within the CM in the manner that is most efficient for your programming needs.
- Front-end/CM communication operators such as ***sum** and **array-to-pvar** let you quickly move large amounts of data between the front end and the CM.
- Data transformation operators such as **scan!!**, **spread!!**, and **sort!!** allow you to design just the kinds of data transformations you need.
- Configuration operators such as ***cold-boot**, **def-vp-set**, and **with-vp-set** give you control of the shape of the current processor grid, and, in so doing, let you specify the size and shape of your pvars.

That's all the main parallel programming tools of *Lisp in a nutshell. In the next chapter, we'll look at the error-handling features of *Lisp, and show how you can use the Lisp debugger to examine and diagnose bugs you encounter while writing *Lisp programs.

"Oh, don't the days seem lank and long,
When all goes right and nothing goes wrong..."
W. S. Gilbert

Chapter 4

When Things Go Wrong

No introductory guide can account for the unexpected. Even if you follow the instructions of this document to the letter, typing in everything exactly as shown, you may still receive unexpected warnings and error messages, or suddenly find yourself facing an imposing display of debugger information. The purpose of this chapter is to show you the types of warnings and errors that you can expect to see, and to show how you can deal with them.

Note: The warning messages and debugger examples shown below are taken from the Sun Common Lisp version of *Lisp. Other versions of *Lisp may have different methods of displaying and handling errors, but the essential points will still apply. Consult your Lisp environment's documentation for more information on the debugger and error-checking features it provides for you.

4.1 Warning Messages

Warnings take many forms. Some are messages from the Lisp system that you're using, such as this one, which tells you the Lisp garbage collector is active:

```
;;; GC: 174988 words [699952 bytes] of dynamic storage in use.
;;; 644210 words [2576840 bytes] of free storage available.
```

For the Curious: If you prefer not see such warning messages, many Lisp systems will let you turn them off by setting an appropriate global variable. For example, in Sun Common Lisp you can turn off GC warnings by typing **(setq lcl::*gc-silence* t)**. For Lucid Common Lisp on VAX front ends, the corresponding command is **(setq sys::*gc-silence* t)**.

Other warnings come from *Lisp itself. Most of the *Lisp warnings that you're likely to see will come from the *Lisp compiler. Depending on the settings of certain *Lisp compiler variables, you may see a warning when you compile a function. For example:

```
> (defun simple (x) (+!! x 2))
> (compile 'simple)
;;; Warning: *Lisp Compiler: While compiling function SIMPLE:
;;;          The expression (+!! X 2) is not compiled because
;;;          the compiler cannot find a declaration for X.
```

For now, you can pretty much ignore such warnings. These messages are simply telling you that the *Lisp compiler lacks information that it requires to fully compile a piece of *Lisp code. Your functions are still being compiled, though perhaps not as efficiently as possible. (We'll return to this in more detail when we look at the *Lisp compiler in Chapter 5.)

4.2 Error Messages

Errors are just as varied as warnings, and range from simple typing mistakes, such as

```
> (pront 'oops!)
>>Error: The function PRONT is undefined.
SYMBOL-FUNCTION:
   Required arg 0 (S): PRONT
:C  0: Try evaluating #'PRONT again
:A  1: Abort to Lisp Top Level
->
```

to more detailed messages triggered by supplying incorrect arguments to a function:

```
> (/!! 3 0)
>>Error: In interpreted /!!.
   The result of a (two argument) float /!! overflowed.
   There are 8192 selected processors,
   8192 processors have an error.
   A SINGLE-FLOAT temporary pvar stored at location 458752
   caused the error.
*LISP-I::/!!-2:
...
:A  : Abort to Lisp Top Level
->
```

4.2.1 Recovering from Errors

While you're learning to use *Lisp, you'll probably just want to exit from the debugger and get back to the *Lisp prompt immediately. There is usually a simple command that you can type to abort from the debugger and get back to the top-level prompt. In the Lucid version of *Lisp, the command is

```
-> :A
Abort to Lisp Top Level
>
```

When You Abort, Remember to (*warm-boot)

But simply aborting from the debugger is not enough. Remember, you're programming on two machines: your front end and the CM. Aborting from the debugger resets your front end and returns you to the top-level Lisp prompt, but it does *not* automatically reset the state of the CM as well. To do this, you must also call the *Lisp operator

```
> (*warm-boot)
```

The ***warm-boot** command resets the internal state of the CM and performs other minor housecleaning operations that return both front end and CM to a normal, ready-to-run state.

You should get into the habit of calling ***warm-boot** whenever you abort from the debugger, because otherwise the CM may be left in a confused state that will prevent your code from running correctly. Even worse, you may get spurious error messages from otherwise error-free code; these errors can be very hard to trace, precisely because the error lies not in your code, but in the state of the CM. As an example of what can happen, try this:

```
> (*sum 1)   ;; NOTE: This example was run on an 8K CM
8192

> (*when (evenp!! (self-address!!)) (break))
>>Break: break
EVAL:
 Required arg 0 (EXPRESSION): (*WHEN (EVENP!! (SELF...
:C 0: Return from Break
:A 1: Abort to Lisp Top Level
-> :a
Abort to Lisp Top Level

> (*sum 1)
4096
```

Here we've used the Common Lisp function **break** to simulate an error within the body of the ***when** form. The ***when** in this example deselects half of the processors. When you abort from the debugger, these processors remain deselected, as the second call to ***sum** shows.

This is easily corrected by calling ***warm-boot**:

```
> (*warm-boot)
> (*sum 1)
8192
```

Whenever you encounter a bizarre inconsistency of this sort while running your code, it's a good idea to try calling ***warm-boot** to reset the CM, and then try running your code again.

For More Persistent Errors, Try (*cold-boot)

Occasionally, you'll encounter an error message that persists even after you've called ***warm-boot**. For these sorts of persistent errors, you can try calling

```
> (*cold-boot)
```

This will reinitialize both *Lisp and the CM, performing a much more thorough housecleaning than ***warm-boot**. If the error is related to permanent pvars—which are reallocated when you call ***cold-boot**—you may want to try typing

```
(*cold-boot :undefine-all t)
```

The **:undefine-all** argument to ***cold-boot** wipes out *every* pvar and VP set in the *Lisp environment, leaving you with a clear field in which to work. (By the way, this will also wipe out any pvars and VP sets defined by your *Lisp programs—you will probably need to reload your *Lisp code to have it run correctly.)

If an obscure error persists even after this, you may want to ask your systems administrator or applications engineer to help you diagnose the problem.

4.3 *Lisp Error Checking

*Lisp provides a global variable, ***interpreter-safety***, that controls the amount of error checking that is performed by the *Lisp interpreter. This affects the kinds of error messages

that you see when you run your code as well as the speed with which your interpreted code will run. (There are similar variables for the *Lisp compiler; we'll look at them in Chapter 5.)

The value of ***interpreter-safety*** must be an integer between 0 and 3. The effects of each of these values are:

0	Most run-time error checking is disabled.
1	Minimal run-time error checking is performed, and an error may not be signaled at the exact point in your code at which it occurred.
2	This setting is reserved for future expansion; don't use it.
3	Full run-time error checking. Errors are signaled immediately.

The default value of ***interpreter-safety*** is 3, but you can set ***interpreter-safety*** yourself to select the level of error checking that you prefer. For example, to turn off interpreter error checking, type

```
> (setq  *interpreter-safety* 3)
```

Roughly, 3 means "full error checking," while 0 means "no error checking." This means that you should run your code with an ***interpreter-safety*** level of 3 while debugging, and at an ***interpreter-safety*** of 0 once it is running correctly to get the best speed from your interpreted code. (**Note:** To obtain the fullest possible speed from your code, you'll want to declare and compile it, as described in Chapter 5.)

An ***interpreter-safety*** level of 1 is useful mainly for testing nearly completed code, because when errors occur you are given an indication of the *fact* that they occurred, but your code is not slowed down to the point where the exact *location* of the error can be indicated. At this safety level, an error detected on the CM will only be reported the next time you try to read a value from the CM. For example:

```
> (setq *interpreter-safety* 1)

> (/!! 3 0) ;; The error occurs here, but is not signaled...
#<FLOAT-Pvar 5-32 *DEFAULT-VP-SET* (32 16)>

> (+!! 2 3) ;; Still not reading a value from the CM...
#<FIELD-Pvar 11-3 *DEFAULT-VP-SET* (32 16)>

> (*sum 1)  ;; Upon reading a value, the error is signaled:
>>Error: The result of a (two argument) float /!! overflowed
CMI::WAIT-UNTIL-FEBI-OUTPUT-FIFO-NOT-EMPTY:
:A  0: Abort to Lisp Top Level
->
```

4.4 Common Errors

Now let's take a quick look at some common errors that you may encounter while learning to use *Lisp. This is by no means an exhaustive list, but it should give you an idea of how to tackle the errors that you do encounter.

4.4.1 Executing *Lisp Code without Having a CM Attached

Unless you're using the *Lisp simulator, you *must* have a CM attached in order to execute *Lisp code. It's easy to tell when you're not attached to a CM: you'll find yourself getting "bus errors" from perfectly legal *Lisp code.

For example:

```
> (cm:detach) ;; Detach from the CM
> (+!! 2 3 4)
>>Trap: Interrupt: bus error
CMI::PTR-REF-LOW:
   Required arg 0 (PTR): NIL
:A Abort to Lisp Top Level
-> (cm:attached)
NIL
NIL
```

As shown here, you can call the Paris function **cm:attached** to determine whether a CM is currently attached. (See the discussion of **cm:attached** in Appendix C.)

Another common error is caused by attaching to a CM, but neglecting to call ***cold-boot**. Until you call ***cold-boot** for the first time, you will not be able to execute your *Lisp code.

```
> (cm:attach)
8192
> (+!! 2 3 4)
>>Error: Timed out on safe FIFO read.
CMI::WAIT-UNTIL-FEBI-OUTPUT-FIFO-NOT-EMPTY:
:A 0: Abort to Lisp Top Level
->
```

Both types of errors are easily corrected by calling ***cold-boot**. The ***cold-boot** operator automatically attaches you to a CM if you're not already attached, and it initializes both *Lisp and the CM so that you can begin running code on the machine.

4.4.2 Running Out of CM Memory

There are two kinds of CM memory: the stack and the heap. The heap is used for permanent pvars, and the stack is used for local and temporary pvars. If you allocate so many pvars in either kind of memory that there isn't room for more, you'll get an error.

Running Out of Stack Memory

For example, stack pvars are allocated by ***let**, ***let***, and by nearly all the functions in *Lisp that return a pvar. You're not likely to run out of stack memory by using either ***let** or ***let***, because these operators automatically deallocate any stack pvars they create. However, if you call to a *Lisp function within a loop, you can easily find yourself running out of stack memory:

```
> (dotimes (i 10000) (+!! 3.14 2.17 9.99))
>>Error: You have run out of CM memory
   while trying to allocate 32 bits of stack.
CMI::TRY-TO-GET-MORE-CM-STACK-MEMORY:
   Required arg 0 (NUMBER-OF-BITS-TO-ALLOCATE): 32
:A 0: Abort to Lisp Top Level
```

Each time the **+!!** call is executed, it allocates a temporary pvar on the stack. These temporary pvars steadily pile up on the stack until there is no more stack space left. To recover from this kind of error, abort from the debugger and call ***warm-boot**, which will clear out the stack.

For the Curious: An easy way to prevent this kind of error is to make sure that all your calls to *Lisp functions are made within *Lisp operators that automatically clear the stack after they have finished executing, for instance ***set** and ***let**. For example, try:

```
> (*defvar temp-pvar 0.0)
> (dotimes (i 10000) (*set temp-pvar (+!! 3.14 2.17 9.99)))
NIL
```

*Lisp code that you write will typically be structured so that this is the case—that is, all pvar computations will take place within the scope of *Lisp operators that clear the stack. However, if you find yourself running out of stack space, you should check your code to make sure you are not repeatedly allocating stack pvars that never get cleared away.

A complete list of *Lisp operators that automatically clear the stack can be found in the entry for ***defun** in the **Lisp Dictionary*.

A stack-related resource you may find yourself running short of is temporary pvars:

```
> (dotimes (i 1000) (+!! 0 1 2))
>>Error: You have run out of temporary Pvars.
   You must now do a *COLD-BOOT.
*LISP-I::POP-LOCAL-ERROR:
:A  0: Abort to Lisp Top Level
->
```

When you get an error like this, you haven't necessarily run out of memory, but you *have* run out of the data structures used to keep track of the stack memory you allocate. At this point, you *may* be able to get away with simply calling ***warm-boot** to restore the pool of free temporary pvars, but if not you can always call ***cold-boot** to recover from this error. (In the same way, you can find yourself running short of VP set and geometry data objects—the same solution applies in these cases.)

Running Out of Heap Memory

Heap pvars are allocated by the operations ***defvar** and **allocate!!**. You're not likely to exhaust heap memory by using ***defvar**, but the main purpose of **allocate!!** is to allow you to rapidly allocate as much of the heap as you want for pvars. You can easily use up all the available heap memory by calling **allocate!!** indiscriminately:

```
> (defvar pvar-list nil)

> (loop (push (allocate!! 3.14159) pvar-list))
>>Error: You ran out of CM memory
   while trying to allocate 32 bits on the heap.
CMI::ALLOCATE-HEAP-FIELD-ADDRESS-AND-CM-MEMORY:
   Required arg 0 (NUMBER-OF-BITS-TO-ALLOCATE): 32
:A 0: Abort to Lisp Top Level
->
```

The easiest way to recover from heap memory errors is to call ***cold-boot**, which destroys all pvars created by **allocate!!**, and then reallocates pvars you have defined by ***defvar**.

Another way to recover is to use ***deallocate** to remove some of the heap pvars you have defined. In the above example, the allocated heap pvars are stored in a list, so it is easy to deallocate them:

```
> (dolist (pvar pvar-list) (*deallocate pvar) (pop pvar-list))
NIL
```

Tracing Stack Memory Use

Even if your program doesn't contain simple loops like those shown above, it may still run out of memory if it allocates a large number of pvars, or allocates pvars that contain very large data structures. In particular, if you get an "out of stack memory" error, you'll want to check your program to see where it is allocating the most stack memory.

*Lisp includes a function, **trace-stack**, that you can use to trace the stack memory usage of your program, and thereby determine what parts of your code are using the most stack memory. For example, let's trace the function

```
> (defun trace-test (a b c)
     (*!! a (+!! b c)))
```

We can use **trace-stack** to trace the amount of stack memory used by this function:

```
> (trace-stack :init)
Stack tracing is now on in :TRACE mode.
Current stack level is 1536.
Maximum stack limit is 1536.
1536
1536
> (trace-test 9 3 2)
#<FIELD-Pvar 9-7 *DEFAULT-VP-SET* (128 64)>

> (trace-stack :reset)
Stack tracing is now on in :BREAK mode.
Current stack level is 1536.
Maximum stack limit is 1554.
1536
1554
```

This records the amount of stack memory used by the call to **trace-test**. We can then use **trace-stack** to cause a continuable error whenever stack usage reaches or exceeds this limit:

```
> (trace-test 9 3 2)
>>Error: Stack has reached/exceeded traced maximum of 1554.
         Stack is now at 1554.
*LISP-I::MAX-STACK-LEVEL-CHECK:
:C  0: Continue until next stack increase.
:A  1: Abort to Lisp Top Level
-> :a
Abort to Lisp Top Level
Back to Lisp Top Level
```

And when we're finished with the stack tracing facility, we can switch it off:

```
> (trace-stack :off)
Stack tracing is now off.
Current stack level is 1554.
Maximum stack limit is 1554
1554
1554
```

For more information, see the entry for **trace-stack** in the **Lisp Dictionary.*

Displaying CM Memory Use

*Lisp also includes ***room**, a parallel equivalent of Common Lisp's **room** function. You can use ***room** to get a general description of CM memory use:

```
> (*room)
*Lisp system memory utilization

Vp Set *DEFAULT-VP-SET*, (32 16)
  Stack memory usage    : 11
  Heap memory usage     : 0
  *Defvar memory usage : 8
  Overhead              : 10
  Total for Vp Set      : 29

Overall totals
Stack memory usage    : 11
Heap memory usage     : 0
*Defvar memory usage : 8
Overhead              : 18
Total                 : 37

11
0
8
18
```

The ***room** function is also described in more detail in the **Lisp Dictionary.*

4.4.3 Functions That Don't Promote Scalars to Pvars

Most *Lisp operations that accept pvars as arguments also accept scalars of the same type, and automatically promote them to pvars. There are a few functions, however, that don't. When one of these functions encounters a scalar value where it expects a pvar, it will warn you that the scalar argument should be a pvar:

```
> (describe-pvar 3)
>>Error: 3 is not a Pvar.
DESCRIBE-PVAR:
   Required arg 0 (PVAR): 3
   Optional arg 1 (STREAM): #<Stream SYNONYM-STREAM 30B2BE6>
:A 0: Abort to Lisp Top Level
```

You'll also get an error of this sort if you supply a scalar value of an incorrect type. For example, arithmetic operations such as **+!!** only coerce numeric scalars to pvars:

```
> (+!! #\c 3)
>> Error: An argument to +!!, having value #\c,
   is not a pvar, but should be
*LISP-I::NEW-PVAR-CHECK:
   Required arg 0 (PVAR): #\c
   Required arg 1 (FUNCTION-NAME): +!!
:C  0: Test the assertion again
:A  1: Abort to Lisp Top Level
```

You can recover from these errors simply by aborting from the debugger, calling ***warm-boot**, and then calling the function again with a proper pvar argument.

This kind of error can also occur when you have compiled a function with a declaration restricting its arguments to pvars. Scalar arguments passed to these compiled functions will *not* be promoted. For example:

```
> (defun test (a)
   (declare (type single-float-pvar a))
   (+!! a a))
> (compile 'test)

> (test 3)
>>Trap: Interrupt: bus error
TEST:
   Required arg 0 (A): 3
:A 0: Abort to Lisp Top Level
->
```

4.4.4 Obscure Hardware and Software Errors

Some error messages, caused either by unusual CM states or real hardware errors, are downright impenetrable for the average user:

```
>>Error: Detected CM Exception while waiting for data from CM.
The following status message might help identify the problem.
   There are 28. CM and/or SPRINT chips reporting errors.
   This usually indicates an illegal memory reference.
   The cause could be an invalid field-id or incorrect
   length argument in a PARIS instruction.
CMI::WAIT-UNTIL-FEBI-OUTPUT-FIFO-NOT-EMPTY:
...
<Half a page of continuation options>
...
:A  0: Abort to Lisp Top Level
->
```

Your best bet in handling these kinds of obscure errors is to abort from the debugger, call ***warm-boot** or ***cold-boot**, and try running your code again. If errors of this kind persist, they should be reported to your systems manager or applications engineer for correction.

4.5 Using the Debugger

Once you feel comfortable working with *Lisp, you'll want to take a closer look at the debugger to see what features it offers you. Let's take a quick look at a few debugger commands that you may find helpful in debugging your code.

As an specific example, type

```
> (/!! 1.0 (self-address!!))
```

This causes a floating-point overflow due to the division by zero for processor 0.

Assuming an 8K CM, this will producing the following debugger output:

```
>>Error: In interpreted /!!.
   The result of a (two argument) float /!! overflowed.
   There are 8192 selected processors,
   1 processor has an error.
   A pvar of type SINGLE-FLOAT caused the error.
*LISP-I::/!!-2:
   Required arg 0 (A): #<FLOAT-Pvar 3-32 ... >
   Required arg 1 (B): #<FIELD-Pvar 5-13 ... >
:C 0: Ignore error.
   1: Ignore Error.
   2: Display Processors With Error.
   3: Display Value in Processors with Error.
   4: Display Selected Processors.
   5: Display Value in Selected Processors.
   6: Display Value in All Processors.
   7: *Set Value in Processors with Error.
:A 8: Abort to Lisp Top Level
->
```

The debugger display includes:

- the error that occurred (a floating-point overflow)
- the number of processors that are currently selected (8192)
- the number of those processors that signaled the error (1)
- the type of pvar that caused the error (**single–float**)
- the name of the internal function in which the error occurred (***lisp–i::/!!–2**)
- the arguments to that function (a **float** pvar and a **field** pvar)
- a list of debugging actions, each with an associated command to invoke it

At this point, there are a number of things that you can do:

- You can type the debugger command **:c** to continue, ignoring the overflow error.
- You can type **:b** for a backtrace of the chain of function calls leading to the error:

```
-> :b
*LISP-I::/!!-2 <- EVAL <- SYSTEM:ENTER-TOP-LEVEL
```

- You can type **:n** and **:p** to move up and down the chain of calls:

```
-> :n
EVAL:
     Required arg 0 (EXPRESSION):
          (/!! (!! 1.0) (SELF-ADDRESS!!))
-> :p
*LISP-I::/!!-2:
     Required arg 0 (A): #<FLOAT-Pvar 3-32 ... >
     Required arg 1 (B): #<FIELD-Pvar 5-13 ... >
```

- You can use the supplied debugging options to view the values contained in the processors that signaled the error:

```
-> 3
Display Value in Processors with Error.
    0 = #<FLOAT :PLUS-INFINITY>
```

From this display you see the value in each processor that signaled an error. In this case, there was only one, processor 0, whose send address of 0 caused the division to overflow, returning a result of floating-point infinity.

- And as we have already seen, you can type **:a** to abort back to the Lisp prompt.

```
-> :a
Abort to Lisp Top Level
Back to Lisp Top Level
> (*warm-boot)
NIL
```

Your debugger will have many more commands than there is room to discuss here. You can use the command **:h** from within the debugger to see a list of the possible commands that you can use. Also, consult the documentation for your Lisp development system for more information about the debugger and the tools it offers you for debugging your code.

4.6 Summary: Find That Bug and Step on It!

In this chapter, we've seen

- some of the types of warning and error messages displayed by *Lisp
- the most common error messages you can expect to see, and how to handle them
- the kinds of information displayed by the Lisp debugger
- the tools that the debugger gives you for diagnosing errors
- the basic method for exiting from the debugger: abort and ***warm-boot**
- the ways in which ***warm-boot** and ***cold-boot** help you recover from errors
- the *Lisp variable ***interpreter-safety***, which controls the display of errors by the *Lisp interpreter

Now that we've seen some of the ways in which your code might break down, let's take a look at the tools you can use to make it run like a dream. In the next chapter, we'll look at the *Lisp compiler, and see how you can use it to compile your *Lisp code into fast and efficient Lisp/Paris instructions.

"I have nothing to declare except my genius."
Oscar Wilde

Chapter 5

Declaring and Compiling *Lisp Code

Depending on your interests and inclinations, you may begin writing your *Lisp code with an eye to immediately compiling it, or you may leave the step of declaring and compiling your code until after you have it running in an interpreted form. No matter what your coding style happens to be, you'll need to declare your code properly to have it compile completely. In this chapter we'll look both at the *Lisp compiler itself, and at the kinds of type declarations that are used in *Lisp.

You'll also want to refer to Chapter 4, "*Lisp Type Declaration," of the **Lisp Dictionary*, which contains a complete and detailed set of guidelines for properly declaring your code.

5.1 The *Lisp Compiler

The *Lisp compiler is an extension of the Common Lisp compiler, and is invoked automatically whenever you compile a section of Lisp code.

*Lisp functions and macros are compiled exactly as they are in Common Lisp:

- by calling the Common Lisp function **compile** to compile a specific function:

```
(compile 'add-mult!!)
```

- by calling the Common Lisp function **compile-file** to compile a file of code:

```
(compile-file "starlisp-code.lisp")
```

- by whatever method you usually use to compile Lisp code.

  Depending on the Lisp programming tools your editor provides, there may be a special keystroke you can use to compile a function definition from within the editor. In Emacs-style editors this keystroke is commonly Ctrl-Shift-C or Meta-Shift-C. Check the manual for your editor for more information.

Just as in Common Lisp, it is not necessary to compile your *Lisp code before you run it, because uncompiled code is automatically handled by the *Lisp interpreter. However, you won't see the best performance from your code if you don't compile.

5.1.1 What the *Lisp Compiler Does

The *Lisp compiler is different from most standard compilers. It doesn't turn your *Lisp code directly into machine-language instructions. Instead, the *Lisp compiler converts your code into a mixture of Common Lisp code and calls to Paris, the low-level programming language of the CM. This code is then passed through the Common Lisp compiler to be converted into machine code.

*Lisp is not a strongly typed language, but Paris *is*. Thus, for the *Lisp compiler to convert your *Lisp code into Lisp/Paris instructions, you must supply a complete type declaration for each parallel variable, function argument, and returned value in your code. You must also provide declarations for Common Lisp variables and expressions that are used within *Lisp expressions.

You should take the time to declare your *Lisp code so that it can be compiled completely for a number of reasons:

- Compiled *Lisp code executes much faster than interpreted *Lisp code.
- Compiled *Lisp code is more efficient:
  - it uses less CM memory for temporary variables
  - it takes advantage of specialized Paris operations to combine a number of *Lisp operations into a single Paris instruction
  - it eliminates the type-checking overhead of interpreted *Lisp code

5.2 Compiler Options

The *Lisp compiler has many options and parameters that you can use to control the way your code is compiled. This section describes two compiler parameters, ***warning-level*** and ***safety***, that will be very important to you in learning to use the compiler.

5.2.1 Compiler Warning Level

You can instruct the *Lisp compiler to warn you if it encounters a section of *Lisp code that it cannot fully compile, because of missing declarations or other reasons. The value of the global variable ***warning-level*** determines the kind and number of warnings that the compiler displays. The legal values for ***warning-level*** are:

:high	Detailed warnings are displayed whenever code cannot be fully compiled.
:normal	Warnings are only displayed for inconsistencies in type declarations, as when arguments to a function are not of the correct type.
:none	No warning messages are displayed.

The default value for ***warning-level*** is **:normal**, but while you're learning to use the compiler, it is better to leave it set it to **:high**:

```
> (setq *warning-level* :high)
```

5.2.2 Compiler Safety Level

You can also instruct the *Lisp compiler to insert error-checking statements in the code that it produces. The value of the global variable ***safety*** determines the amount of error-checking code that is added, and must be an integer from 0 to 3. The effects of each of these values are:

0	No error-checking code is included.
1	Limited error-checking code is included, so an error may not be signaled at the exact point in your code at which it occurred.
2	Run-time safety control. See description of ***immediate-error-if-location*** below.
3	Full error-checking code is included, so that an error will always be signaled at the exact point in your code at which it occurred.

Roughly speaking, the higher the value of ***safety***, the more error-checking code is included. This means that you should compile your code at a ***safety*** level of 3 while debugging, and compile at a ***safety*** level of 0 when you want to execute your code at full speed. Just as with ***warning-level***, you can change the value of ***safety*** by using **setq**:

```
> (setq *safety* 3)   ;;; Full safety level
```

When your *Lisp code is compiled at a ***safety*** level of 2, you can choose *at run time* between the effects of levels 1 and 3. This allows you to choose the level of error checking that you want without the need to recompile your code. The effect of ***safety*** level 2 is controlled by the global variable ***immediate-error-if-location***.

If ***immediate-error-if-location*** is:

t Errors are signaled immediately, as with ***safety*** level 3.
nil Errors may not be signaled immediately, as with ***safety*** level 1.

You can change the value of ***immediate-error-if-location*** at any time:

```
> (setq *immediate-error-if-location* t)   ;;; Full safety
```

5.2.3 Other Compiler Options

Along with ***safety*** and ***warning-level***, there are a number of other compiler options that you can modify to control the manner in which your *Lisp code is compiled. Most of these options can safely be ignored for right now, but you will want to know how to examine and change them. The function

```
(compiler-options)
```

displays a menu of the current compiler options, along with their settings. You can then modify each of these settings interactively. You can also change any of the compiler's options by changing the value of a global variable (as shown for ***warning-level*** and ***safety*** above).

For the Curious: The full set of compiler options is described in Chapter 5, "*Lisp Compiler Options," of the **Lisp Dictionary*.

5.3 Displaying Compiled Code

One of the best ways to see the effect of the *Lisp compiler on your code is to compile it in such a way that the Lisp/Paris form of the code is displayed.

Displaying All Compiled Code

The simplest way to do this is by typing

```
> (setq slc::*show-expanded-code* t)
```

The global variable **slc::*show-expanded-code*** controls whether or not the *Lisp compiler automatically displays each piece of Lisp/Paris code that it generates. For example, with **slc::*show-expanded-code*** set to **t**, when you compile the function

```
> (defun test (pvar1 pvar2 pvar3)
    (declare (type single-float-pvar pvar1 pvar2 pvar3))
    (*!! (+!! pvar1 pvar2) pvar3))
```

the following expanded code is displayed:

```
(LET* ((SLC::STACK-FIELD (CM:ALLOCATE-STACK-FIELD 32))
       (#:+*!!-INDEX-2 (+ SLC::STACK-FIELD 32)))
   (DECLARE (TYPE SLC::CM-ADDRESS
                  SLC::STACK-FIELD #:+*!!-INDEX-2))
   (DECLARE (IGNORE #:+*!!-INDEX-2))

   ;; (*!! (+!! pvar1 pvar2) pvar3).
   (CM:F-ADD-MULT-1L SLC::STACK-FIELD
      (PVAR-LOCATION PVAR1) (PVAR-LOCATION PVAR2)
      (PVAR-LOCATION PVAR3) 23 8)

   (SLC::ALLOCATE-TEMP-PVAR
      :TYPE :FLOAT :LENGTH 32 :BASE SLC::STACK-FIELD
      :MANTISSA-LENGTH 23 :EXPONENT-LENGTH 8))
```

The important code to notice here is the commented center section—I've set it off with extra space to make it easier to read. As you can see, the *Lisp compiler was able to compress the ***!!** and **+!!** operations into a single step by using the Paris floating-point add/multiply instruction **cm:f-add-mult-1l**. (The rest of the code handles bookkeeping details such as reserving stack space and defining the temporary pvar that will be returned.)

Displaying Specific Compiled Code

If you just want to see the expansion of a particular section of code, the **slc::*show-expanded-code*** variable may not suit your purposes. To examine the expanded form of a particular piece of code, you can use two Common Lisp operations, **pprint** and **macroexpand-1**:

```
> (setq slc::*show-expanded-code* nil)
> (let ((*compiling* t))
    (pprint (macroexpand-1 '(*set (the single-float-pvar a)
                                  (the single-float-pvar b)))))
(PROGN ;; Move (coerce) source to destination - *set.
   (CM:MOVE (PVAR-LOCATION A)
            (PVAR-LOCATION B)
            32)
   NIL)
```

(In this example I've included **the** type declarations so that the **+!!** form will fully compile.)

There are two things to keep in mind when using **pprint** and the macroexpanding functions to display *Lisp code:

- The *Lisp compiler must be enabled (it is by default). To enable the compiler, type:

  ```
  (setq *compilep* t)
  ```

- The *Lisp compiler will only compile macroexpanded forms when the global variable ***compiling*** is bound to **t**. You should use **let** to temporarily bind this variable, as in the example above.

Pretty Print Macroexpand — ppme

Typing the entire **(let ((*compiling* t)) (pprint (macroexpand-1 ...)))** expression around every piece of code you that wish to compile is a clumsy way to view your compiled code. For this reason, the compiler includes a macro that you can use to display the expanded form of a piece of code. Called **ppme** (short for "pretty print macroexpand"), it essentially performs a call to **pprint** and **macroexpand-1**, as in the above example.

You can call **ppme** with almost any piece of *Lisp code. For example:

```
> (ppme (*set (the single-float-pvar a)
              (+!! (the single-float-pvar b)
                   (the single-float-pvar c))))
```

The resulting compiled code looks like this:

```
(progn
  ;; (*set (the single-float-pvar a) (+!! (the single-fl...
  (cm:f-add-3-11 (pvar-location a)
                 (pvar-location b)
                 (pvar-location c)
                 23
                 8)
  nil)
```

There is one limitation, however. The **ppme** macro only expands a piece of code when the outermost operator of the code is a macro. To expand other *Lisp expressions, such as

```
(+!!  (the single-float-pvar b)
      (the single-float-pvar c))
```

enclose them in a *Lisp macro such as ***set**, as shown in the example above.

Depending on the features of your front end's Lisp programming environment, there may be other tools that you can use to view expanded code. In general, any tool that macroexpands a section of code may be used to view the compiled form of *Lisp code (so long as the outermost form of the code is a macro, of course). For example, on Symbolics front ends the editor includes the commands **Macro Expand Expression (Control-Shift-M)** and **Macro Expand Expression All (Meta-Shift-M)** which are used for expanding macros; these tools will also allow you to view the code generated by the *Lisp compiler.

5.4 *Lisp Data Types and Declarations

Just as in Common Lisp, declarations are optional in *Lisp; you don't need to provide type declarations to get your *Lisp code to run. However, if you want to compile your *Lisp code you *must* provide complete declarations for all parallel variables and functions in your program. In this section, we'll take a quick look at *Lisp data types and type specifiers, and then show how you go about including declarations in your programs.

For Inquiring Minds: Chapter 4 of the **Lisp Dictionary* provides a detailed discussion of type declaration in *Lisp, and includes an exact set of rules for determining what declarations you must provide to get your *Lisp code to compile.

5.4.1 Pvar Type Specifiers — (pvar *typespec*)

*Lisp includes all the front-end data types of Common Lisp, and thus also includes the standard type specifiers for those types. However, as we've seen not all front-end data types have pvar equivalents. This section shows you examples of legal type specifiers for each of the pvar data types of *Lisp.

Type specifiers for pvars are typically of the form

```
( pvar typespec )
```

where *typespec* is a type specifier for one of a specific set of scalar data types.

For each of the pvar data types listed below, I've included the basic type specifier for pvars of that type, as well as additional "shorthand" specifiers. (The symbol <=> indicates equivalent forms.) These shorthand specifiers can be used in place of the longer **(pvar ...)** forms wherever a type specifier is required.

boolean — Either **t** or **nil** for each processor.

```
(pvar boolean)
boolean-pvar
```

unsigned-byte — A non-negative integer for each processor.

```
(pvar (unsigned-byte length))
(unsigned-pvar length)
(field-pvar length)
unsigned-byte-pvar
```

signed-byte — A signed integer for each processor.

```
(pvar (signed-byte length))
(signed-byte-pvar length)
signed-byte-pvar
```

defined-float — A floating-point number for each processor.

```
(pvar (defined-float significand-length exponent-length))
(float-pvar significand-length exponent-length)

single-float-pvar <=>   (pvar (defined-float 23 8))
double-float-pvar <=>  (pvar (defined-float 52 11))
```

complex — A complex number for each processor.

```
(pvar (complex (defined-float significand exponent))
(complex-pvar significand exponent)
single-complex-pvar
                    <=> (pvar (complex (defined-float 23 8))
double-complex-pvar
                    <=> (pvar (complex (defined-float 52 11))
```

character — A Common Lisp character for each processor.

```
(pvar character)
character-pvar

(pvar string-char)
string-char-pvar
```

array — A Common Lisp array for each processor.

```
(pvar (array element-type dimension-list))
(array-pvar element-type dimension-list)

(vector-pvar element-type length)
                    <=>  (pvar (array element-type (length)))
```

structure — A Common Lisp structure object for each processor.

```
(pvar structure-name)
structure-name-pvar
```

Note: *structure–name* must be a structure defined by the *Lisp ***defstruct** macro.

front–end — A reference to a front-end value for each processor.

```
(pvar front-end)
front-end-pvar
```

Note: Pvars of this type are created by the *Lisp **front–end!!** operator.

general — A value of any data type for each processor.

```
(pvar t)
(pvar *)
general-pvar
```

Note: By default, undeclared pvars are assumed to be of type **(pvar *)**.

5.4.2 Using Type Declarations

There are three ways of providing type declarations in *Lisp:

- global type declarations, made with *Lisp's ***proclaim** operator
- local declarations, made with Common Lisp's **declare** operator
- type declarations made "on the fly" with Common Lisp's **the** operator

We'll take a quick look at of each of these declaration operators in this section.

Global Type Declarations — *proclaim

The ***proclaim** operator is much like Common Lisp's **proclaim**, but is used to provide type information to the *Lisp compiler.

The ***proclaim** operator is used to declare the data type of:

- permanent pvars allocated by ***defvar**
- scalar variables used in pvar expressions
- values returned by user-defined functions

Important: You *must* use ***proclaim** to provide global type declarations to the *Lisp compiler, instead of Common Lisp's **proclaim** form. The ***proclaim** operator passes type information directly to the *Lisp compiler; **proclaim** does not.

When used to declare the type of a permanent pvar, ***proclaim** takes the form

```
(*proclaim '(type pvar-type-specifier pvar-name pvar-name ... ))
```

as in the following examples:

```
(*proclaim '(type single-float-pvar float-pvar))
(*defvar float-pvar 3.14)

(*proclaim '(type (pvar (signed-byte 8))  integer-pvar-1
                                          integer-pvar-2))
(*defvar integer-pvar-1 -5)
(*defvar integer-pvar-2 26)
```

Similarly, when used to declare the type of scalar variables used in pvar expressions, ***proclaim** takes the form

```
(*proclaim '(type scalar-type-specifier var-name var-name ... ))
```

as in:

```
(*proclaim '(type single-float account-balance))
(defvar account-balance 100) ;; Scalar variable, not a pvar
(*set interest (*!! account-balance .05))

(*proclaim '(type complex point1 point2))
(defvar point1 #C(1.0 2.0))
(defvar point2 #C(4.0 5.0))
(*set total-real-length
   (+!! (realpart!! point1) (realpart!! point2)))
```

When used to declare the returned type of a user-defined function, ***proclaim** takes the form

```
(*proclaim
  '(ftype (function argument-types returned-value-type) function-name))
```

as in:

```
(*proclaim '(ftype (function (pvar pvar) (pvar single-float))
               average!!))
(defun average!! (pvar1 pvar2) (/!! (+!! pvar1 pvar2) 2))
```

Note: A handy rule of thumb for using ***proclaim** is that every correct ***proclaim** definition ends with at least two closing parentheses. If you find yourself closing a ***proclaim** with only one parenthesis, it's probably because your type specifier isn't correctly formed. For example, the declaration

```
(*proclaim '(pvar (unsigned-byte 23)) any-pvar) ; Wrong
```

is incorrect because it lacks the enclosing **(type ...)** around the type specifier and the pvar name. The correct form is:

```
(*proclaim '(type (pvar (unsigned-byte 23)) any-pvar)) ; Right
```

Local Type Declarations — declare

For local declarations within your *Lisp code, use Common Lisp's **declare** operator.

The **declare** operator is used to declare the data type of:

- local pvars created by ***let** and ***let***
- arguments to user-defined functions
- local variables used in pvar expressions

When used to declare the type of a local pvar, **declare** takes the form

```
(declare (type pvar-type-specifier pvar-name pvar-name ... ))
```

as in:

```
(*let ((float-pvar (random!! 1.0))
       (field-pvar (random!! 10)))
   (declare (type single-float-pvar float-pvar))
   (declare (type (field-pvar 8) field-pvar))
   (+!! float-pvar field-pvar))
```

When used to declare the types of arguments to user-defined functions, **declare** resembles the argument type declarations used in other computer languages:

```
(defun pvar-function (pvar1 pvar2)
   (declare (type single-complex-pvar pvar1))
   (declare (type (signed-byte-pvar 16) pvar2))
   (*!! pvar1 pvar2))
```

The **declare** operator is also recognized by *Lisp in all the locations that Common Lisp permits, so you can use it to declare the data type of local variables used in pvar expressions. For example, you can **declare** the type of a looping variable, as in:

```
(do ((i 0 (+ i 2)))
    ((>= i 10))
   (declare (type fixnum i))
   (*set sum-pvar (+!! sum-pvar i)))
```

Note: The loop operator **dotimes** is an exception. Since the index variable of a **dotimes** loop is always an unsigned integer, the *Lisp compiler automatically declares it as such. You do not have to **declare** the data type of a **dotimes** index variable.

In-line Type Declaration— the

You can also use the Common Lisp **the** operator to make in-line declarations in your code, for cases where neither ***proclaim** or **declare** can be used. (As, for example, when you need to declare the data type returned by a Common Lisp expression.)

The **the** operator has the following form:

```
(the type-specifier expression)
```

as in

```
(*set data-pvar
   (the (unsigned-byte 32) (+ normal-limit extra-limit)))
```

5.5 Type Declaration and the Compiler

There are a number of basic principles governing the use and effects of type declarations:

- A type declaration is a promise made by you to the *Lisp compiler, guaranteeing that variables and functions declared to be of a specific type will never have values of any other data type.

  For example, the **declare** form in

  ```
  (defun max-float (pvar1)
       (declare (type double-float-pvar) pvar1)
       (*max pvar1))
  ```

  promises that **max-float** will always be called with a double-float **pvar1** argument.

- The *Lisp compiler uses this information only to turn your code into more efficient Paris instructions. Type declarations don't change the semantics of your code. In particular, type declarations do *not* cause the *Lisp compiler to perform coercions.

  For example, the **the** form in

  ```
  (the single-float-pvar any-pvar)
  ```

  does not automatically convert **any-pvar** into a **single-float-pvar**.

- You must include any needed conversions yourself, via operators such as **coerce!!** as in

```
(the single-float-pvar
     (coerce!! any-pvar 'single-float-pvar))
```

- It is an error for a *Lisp program to violate its own declarations. If your code doesn't match your declarations, the results you obtain will be unpredictable.

- Changing a declaration in your source code does not affect the compiled form of your code. If you change a declaration, you must recompile any code that depends on that declaration to see the effect of the change.

- Length expressions in pvar type declarations are compiled in such a way that they are evaluated at run time. This means you can use declarations such as

```
(pvar (unsigned-byte byte-size))
```

  where the symbol **byte-size** is a variable that determines the byte size used throughout your code. Using evaluated expressions in this fashion is, of course, not as efficient as using constant-sized declarations.

  Warning: This is a *Lisp extension to Common Lisp, and is valid only within pvar declarations. For example:

```
> (defun foo (x)
     (declare (type (unsigned-byte byte-size) x))
     (print x))
> (compile 'foo)
;;; Warning: Illegal type specifier
;;; in (TYPE (UNSIGNED-BYTE BYTE-SIZE) X)
```

  The **(unsigned-byte byte-size)** declaration signals a warning because Common Lisp does not recognize this kind of type specifier.

5.5.1 *Lisp Code That Won't Fully Compile

Some *Lisp code can not be completely compiled by the *Lisp compiler. This includes:

- *Lisp code with undeclared pvars, functions, or scalar expressions

- *Lisp code that uses general pvars (pvars of type **(pvar t)** or **(pvar *)**)

5.6 Example: Compiling and Timing a *Lisp Function

As an example of the tools we've seen in this chapter, let's take a simple *Lisp function, declare it, compile it, and time it to see how fast it runs.

The function we'll use is:

```
(defun add-mult!! (a b c)
   (*!! (+!! a b) c))
```

Let's assume that we want the **add-mult!!** function to take single-precision floating-point pvars as arguments, and return a single-float pvar as a result. The definition of **add-mult!!** with the appropriate declarations added is

```
> (*proclaim   '(ftype  (function (pvar pvar pvar)
                         (pvar single-float))
               add-mult!!))

> (defun add-mult!! (a b c)
      (declare (type (pvar single-float) a b c))
      (*!! (+!! a b) c))
```

Now we can compile this function and see how its speed improves. We'll use the Paris timing operator **cm:time** to see the difference in speed.

The **cm:time** macro takes a single Lisp form (either *Lisp or Common Lisp) as its argument, and uses the timing facility of the CM to provide an accurate description of the time taken to execute that form.

For Simulator Users: The timing examples in this section apply only to hardware *Lisp users. In particular, the Paris **cm:time** operator described here exists only in the hardware version of *Lisp. You will get an error if you call this function while using the simulator.

For example, we can time a call to the interpreted version of **add-mult!!** like this:

```
> (cm:time (add-mult!! (!! 2.0) (!! 4.0) (!! 7.0)))

Evaluation of (ADD-MULT!! (!! 2.0) (!! 4.0) (!! 7.0)))
took 0.005786 seconds of elapsed time,
during which the CM was active for 0.002159 seconds
or 37.00% of the total elapsed time.
```

Note: This is just an example of the kind of timing results you'll see. Your actual results will vary depending on factors such as the number of users on your front end computer.

Note: The first time you run this example, you may also see some additional messages:

```
Warning: Turning off Paris safety for CM:TIME.
Warning: Turning off *Lisp safety for CM:TIME.
Calibrating CM idle timer...
Calculated CM clock speed = 6.99983 MHz
```

You can safely ignore these warnings. They're simply to inform you that the CM timing facility is initializing itself.

When you're timing your *Lisp code, there are three important numbers to watch for:

- the elapsed time (the total time taken to execute the supplied Lisp form)
- the CM active time (the amount of time the CM was actively processing data)
- the percentage utilization of the CM (what percentage of the elapsed time the CM was active)

In the example above, the CM utilization is 37% of the total elapsed time, which means that the CM was busy for only about a third of the total time it took to execute the Lisp expression.

Of course, the function **add-mult!!** executes so quickly that using **cm:time** on a single function call doesn't give a good sense of how fast the function executes. Let's define a function that calls **add-mult!!** a thousand times, and use that to help time the function itself.

```
> (*proclaim '(type (pvar single-float) my-float-pvar))
> (*defvar my-float-pvar (!! 0.0))
> (defun test-loop ()
    (dotimes (i 1000)
      (*set my-float-pvar
        (add-mult!! (!! 2.0) (!! 4.0) (!! 7.0)))))
```

Now let's time a call to **test-loop**:

```
> (cm:time (test-loop))
Evaluation of (TEST-LOOP) took 4.678420 seconds of elapsed
time, during which the CM was active for 2.059298 seconds
or 44.00% of the total elapsed time.
```

This is the result using interpreted *Lisp code. The total elapsed time was four and a half seconds, with the CM active less than half that time.

Now let's compile **test-loop** and **add-mult!!** and time the resulting compiled functions.

```
> (compile 'test-loop)
> (compile 'add-mult!!)
> (cm:time (test-loop))
Evaluation of (TEST-LOOP) took 2.120245 seconds of elapsed
time, during which the CM was active for 1.785779 seconds
or 84.00% of the total elapsed time.
```

This time it took about two seconds to execute the Lisp expression, with the CM busy more than eighty percent of the time. Obviously, compiled code has advantages both in speed of execution and in more complete utilization of the CM.

5.7 Common Compiler Warning Messages

This section describes the compiler warning messages you're most likely to see as you start using the *Lisp compiler, along with the most common reasons for getting these warnings.

5.7.1 Compiler Can't Find Type Declaration...

By far the most common compiler warning is the "compiler can't find type declaration" message. This warning is signaled when your code lacks a type declaration for a variable or function that is needed to fully compile your code. For example:

```
> (setq *warning-level* :high)
> (defun add-constant (pvar c) (+!! pvar c))
> (compile 'add-constant)
;;; Warning: *Lisp Compiler: While compiling PVAR in function
;;;   ADD-CONSTANT: The expression (+!! PVAR C) is not compiled
;;;   because the Lisp compiler cannot find a type declaration
;;;   for the symbol PVAR.
```

Every pvar in your code must have a type declaration, either by a local declaration operator such as **declare** or **the**, or by the global declaration operator ***proclaim**. But simply declaring all your pvars is not sufficient. You must also declare the type of any scalar variables that are used in your *Lisp code.

For example, if we add a declaration for **pvar** to **add-constant**, as in

```
> (defun add-constant (pvar c)
   (declare (type single-float-pvar pvar))
   (+!! pvar c))
```

The function still will not compile:

```
> (compile 'add-constant)
;;; Warning: *Lisp Compiler: While compiling C in function
;;;   ADD-CONSTANT: The expression (+!! PVAR C) is not compiled
;;;   because the *Lisp compiler cannot find a type declaration
;;;   for the symbol C.
```

We must also add a declaration for the scalar variable **c**:

```
> (defun add-constant (pvar c)
   (declare (type single-float-pvar pvar)
            (type single-float c))
      (+!! pvar c))

> (compile 'add-constant)
ADD-CONSTANT
```

5.7.2 Compiler Can't Determine Type Returned by...

A similar warning is the "compiler can't determine type returned by" message. This warning is signaled when your code lacks a declaration for returned value of a function:

```
> (defun add-mult-constant (pvar c m)
     (declare  (type single-float-pvar pvar)
               (type single-float c m))
         (*!! (add-constant pvar c) m))

> (compile 'add-mult-constant)
;;; Warning: *Lisp Compiler: While compiling
;;;   (ADD-CONSTANT PVAR C) in function ADD-MULT-CONSTANT:
;;;   The expression (*!! (ADD-CONSTANT PVAR C) M) is not
;;;   compiled because the *Lisp compiler cannot determine
;;;   the type of value returned by the expression
;;;   (ADD-CONSTANT PVAR C).
```

Every user-defined function in your code must have a type declaration for its returned value. This is typically provided by a ***proclaim** form:

```
> (*proclaim
   '(ftype (function (pvar t) single-float-pvar) add-constant))
```

With this declaration included, the **add-mult-constant** function will now compile:

```
> (defun add-mult-constant (pvar c m)
      (declare (type single-float-pvar pvar)
               (type single-float c m))
          (*!! (add-constant pvar c) m))
> (compile 'add-mult-constant)
ADD-MULT-CONSTANT
```

5.7.3 Compiler Does Not Compile Special Form...

The *Lisp compiler recognizes and compiles a small set of Common Lisp special forms without any special declarations. These forms include **compiler-let**, **let**, **let*** and **progn**. Other special forms require a type declaration indicating the data type they will return. For example:

```
> (defun choose (pvar cond v1 v2)
   (declare (type single-float-pvar pvar)
            (type boolean cond) (type single-float v1 v2))
      (*set pvar (if cond v1 v2)))

> (compile 'choose)
;;; Warning: *Lisp Compiler: The expression
;;;   (SLC::*2!! (THE (PVAR #) (ADD-CONSTANT PVAR C)) (!! M))
;;;   in function CHOOSE is not compiled because
;;;   The *Lisp compiler does not yet compile the special form
;;;   IF. Adding a type declaration to the expression
;;;   (IF COND V1 V2),
;;;   such as (THE (PVAR SINGLE-FLOAT) (IF COND V1 V2))
;;;   will allow the *Lisp compiler to compile the expression,
;;;   and make this warning go away.
```

Adding a **the** type declaration to the **if** form allows this function to compile:

```
> (defun choose (pvar cond v1 v2)
    (declare (type single-float-pvar pvar)
             (type boolean cond) (type single-float v1 v2))
      (*set pvar (the single-float (if cond v1 v2))))

> (compile 'choose)
CHOOSE
```

5.7.4 Compiler Does Not Understand How to Compile...

Some *Lisp functions will not compile properly if passed pvar arguments that are declared to be of varying size (for example, pvars of type **(defined-float * *)**). For example:

```
> (defun pack-pvar (pvar)
    (declare (type (pvar (defined-float * *)) pvar))
    (*pset :no-collisions pvar pvar (enumerate!!)))

> (compile 'pack-pvar)
;;; Warning: *Lisp Compiler: While compiling PVAR in function
;;;   PACK-PVAR: The expression
;;;      (*LISP-I::*PSET-1 :NO-COLLISIONS PVAR PVAR ...)
;;;   is not compiled because *PSET does not understand how to
;;;   compile pvars with element-type defined-float,
;;;   varying length mutable pvars can not currently be
;;;   compiled correctly.
```

Changing the declaration to a fixed-size type (a type specifier that does not have any *'s) will allow this function to be compiled:

```
> (defun pack-pvar (pvar)
    (declare (type (pvar single-float) pvar))
    (*pset :no-collisions pvar pvar (enumerate!!)))

> (compile 'pack-pvar)
PACK-PVAR
```

5.7.5 Function Expected a... Pvar Argument but Got...

Finally, the compiler will also catch misdeclared arguments, signaling a warning when a function receives arguments of an incorrect type:

```
> (defun char-value (char-pvar)
    (declare (type character-pvar char-pvar))
    (int-char!! char-pvar))

> (compile 'char-value)
;;; Warning: While compiling CHAR-PVAR in function CHAR-VALUE:
;;;   Function INT-CHAR!! expected an integer pvar argument
;;;   but got a character pvar argument.
```

In this case, the function **char-value** is intended to convert a character pvar to an integer pvar, but is written so that it makes a call to **int-char!!**, which expects an integer (**unsigned-byte** or **signed-byte**) pvar argument. Changing this function by substituting the correct **char-int!!** function for **int-char!!** eliminates the warning:

```
> (defun char-value (char-pvar)
    (declare (type character-pvar char-pvar))
    (char-int!! char-pvar))
> (compile 'char-value)
CHAR-VALUE
```

5.8 Summary: The Compiler as a Programming Tool

In this chapter, we've seen that:

- The *Lisp compiler is invoked automatically whenever you compile code.
- The *Lisp compiler generates Lisp/Paris code that is much more efficient than interpreted *Lisp code.
- To fully compile your code, the *Lisp compiler needs type declarations for each parallel variable, function, and macro in your code, as well as for scalar variables used in pvar expressions.
- These type declarations resemble the type declarations of Common Lisp, and there are declaration forms for each of the pvar data types in *Lisp.

- The *Lisp compiler includes a number of options, such as ***safety***, that allow you to control the way your code is compiled.
- The compiler option ***warning-level*** allows you to request that the compiler warn you when it can't locate the declarations it needs to compile a section of code.
- Because the *Lisp compiler macroexpands *Lisp code into Lisp/Paris code, you can use tools such as **macroexpand** to see the Paris code the compiler generates.
- You can use the Paris operator **cm:time** to see the difference in speed between interpreted and compiled code.

That's it for the programming and compiling tools of *Lisp. In the next chapter, we'll look at some sample *Lisp functions that use the tools and techniques we've seen in previous chapters to perform useful (and perhaps surprising) tasks.

"Example is always more efficacious than precept."
Samuel Johnson

Chapter 6

*Lisp Sampler — A Scan in Four Fits

In this chapter, we'll look at four short *Lisp examples that perform simple, useful tasks. To give this chapter some continuity, I'll focus on one of the most unique and powerful tools of *Lisp: the **scan!!** operator (which we looked at briefly in Section 3.5). The **scan!!** operator can be used in a variety of ways, some of which may suprise you if you're new to the techniques of parallel programming. We'll see a number of uses for **scan!!** in the "fits" of code below, many of which can be adapted for use in your own code. Of course, I'll also be using other *Lisp operations, so from these examples you'll be able to get a sense of how you can combine *Lisp operators to perform specific tasks in your programs.

Note: Some of the examples in this chapter are rather long, so you may not want to type them in by hand. The file **/cm/starlisp/interpreter/f6100/starlisp-sampler.lisp** in the *Lisp source code directory contains a copy of all code examples used in this chapter. Ask your systems manager or applications engineer if you need help locating this file.

6.1 Fit the First: Adding Very Large Integers

In *Lisp you can define integer pvars of any size, and add them in parallel:

```
> (pref (+!! 12345678901234567890 87654321098765432109) 0)
99999999999999999999
```

But if you have extremely large integers to add (longer than, say, forty digits), this isn't really an efficient way to do it. There's no rule, however, saying that you *have* to add numbers by storing them one-per-processor. You can divide a very large integer up into its individual bits, and store the bits one-per-processor (see Figure 8). You can then use *Lisp operations to perform a parallel addition of these pvars, using all the CM processors.

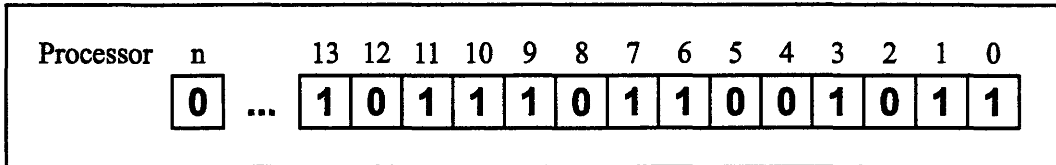


Figure 8. Very large integer stored as a pvar with a single bit value for each processor.

Let's assume that we have two large integers, stored one bit per processor, as described above. This is the same as saying that we have two integer pvars (or, to be more precise, **unsigned-byte** pvars) of length 1, each containing the bits of a very large binary number, one bit per processor. We'll assume that the bits of these pvars are stored in send-address order (that is, with the lowest order bit stored in the processor with a send address of 0).

To add these pvars, we use the rules of binary addition:

- If both source pvars contain 0 for a processor, the result will be 0 for that processor.
- If one source pvar contains 1, and the other 0, the result will be 1.
- If both source pvars contain 1, the result is 0 with a carry of 1.
- A carry value is propagated forward, negating the result for each subsequent processor, until it reaches a processor with 0 in both sources (see Figure 9).

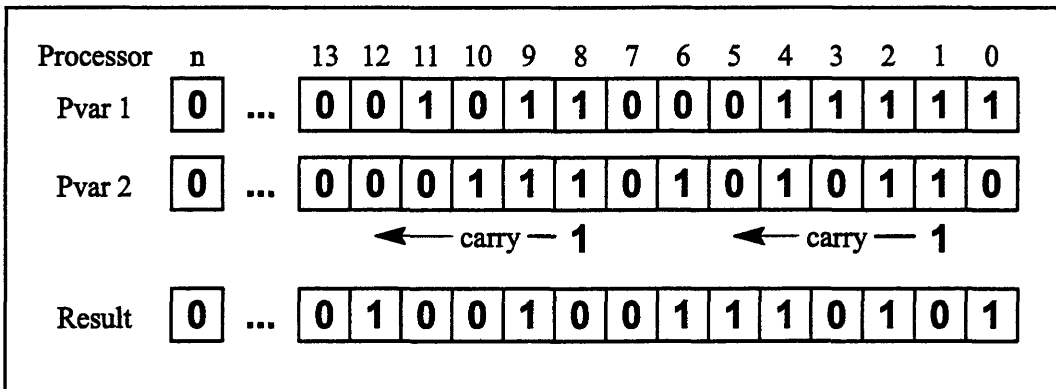


Figure 9. Addition example, showing the propagation of carry bits.

The hardest part of the addition is keeping track of the carry values, and incorporating them into the final result. However, we can easily keep track of them by using the segmented scanning feature of **scan!!** that we saw in Section 3.5.1.

In essence, processors where both sources are 1 and processors where both sources are 0 define "segments" within which carry values are propagated. The carry values start at pairs of 1's and end at pairs of 0's, affecting all values in between. We can use parallel boolean operations to determine where these segments begin and end, and then use a segmented **scan!!** to propagate the effects of the carry bits.

Here's a function that does this:

```
> (defun very-long-add!! (bit-pvar1 bit-pvar2)
    (declare (type (field-pvar 1) bit-pvar1 bit-pvar2))
    (*let  ((zero-zero (=!! 0 bit-pvar1 bit-pvar2))
            (one-one (=!! 1 bit-pvar1 bit-pvar2))
            carry-segments will-receive-carry dest)
      (declare
        (type boolean-pvar zero-zero one-one)
        (type boolean-pvar carry-segments will-receive-carry)
        (type (field-pvar 1) dest))

      ; Determine points at which carries start and end
      (*set carry-segments
        (or!! (zerop!! (self-address!!)) zero-zero one-one))

      ; Determine processors that will be affected by a carry
      (*set will-receive-carry
        (scan!! one-one 'copy!!
           :segment-pvar carry-segments :include-self nil))

      ; Exclude processor zero, because it can't get a carry
      (*setf (pref will-receive-carry 0) nil)

      ; Perform the addition
      (*set dest
       (if!!  (or!! one-one zero-zero)
        ; Pairs of 1's and 0's produce 1 with carry, else 0
        (if!! will-receive-carry 1 0)
        ; All other values will be 0 with carry, 1 otherwise
        (if!! will-receive-carry 0 1)))
      dest))
```

The call to **scan!!** is the real heart of this algorithm. The rest of the function is simply setup code to determine the exact boundaries for the scan, and cleanup code that uses the scan result to calculate the final value for each processor. It's quite common for a function that uses **scan!!** to have this form: setup code, the **scan!!** itself, then cleanup code. We'll see other examples of this later on.

For The Observant: You'll notice that the call to **scan!!** has an **:include-self** argument of **nil**. The effect of this argument is to prevent each processor involved in the scan from including its own value in the result that is passed on to the next processor in the scan. Since we're doing a **copy!!** scan in the **very-long-add!!** function, this has two effects:

- Processors that produce a carry bit (1/1 processors) aren't affected by it.
- Processors that halt propagation of a carry bit (0/0 processors) *are* affected by it.

The **:include-self** argument to **scan!!** is a handy tool for controlling the effects of a scan, because it effectively "shifts" the effect of the scan ahead by one processor. We'll see other examples of the use of **:include-self** later on in this chapter.

Now, since the bit pvars we'll be using have the least significant bit stored in processor 0, we'll want a function that displays these pvars with this bit on the right, rather than on the left, as **ppp** would display them.

Here's a definition for a function that does this, called **pbp** (short for **print-bit-pvar**):

```
> (defun pbp (pvar &key (length 20) title) "Print Bit Pvar"
    (*let ((display-pvar
             (if!! (<!! (self-address!!) length)
                   (pref!! pvar (-!! length (self-address!!) 1))
                   0)))
      (ppp display-pvar :end length :title title))
    (values))
```

The **pbp** function takes a pvar, a length, and a title, and prints the specified number of bit values with the low-order bits on the right. It uses the communication operator **pref!!** to make a copy of the specified part of the pvar with its values "flipped" end for end.

We can define a simple bit pvar with a value of 1 for the first 12 processors, and then use both **ppp** and **pbp** to display the first 20 values of this pvar:

```
> (*defvar bits (if!! (<!! (self-address!!) 12) 1 0))
BITS

> (ppp bits :end 20)      ; low-order bits on left
1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0

> (pbp bits :length 20)  ; low-order bits on right
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1
```

Now, to test **very-long-add!!**, we'll write a function that creates two bit pvars, displays them, and adds them.

```
> (defun test-very-long-add ()
    (let  ((length1 (+ 12 (random 5)))
           (length2 (+ 12 (random 5))))
       (*let ((bit-pvar1 0) (bit-pvar2 0))
          (declare (type (field-pvar 1) bit-pvar1 bit-pvar2))

          ; Store random binary numbers in the bit pvars
          (*when (<!! (self-address!!) length1)
             (*set bit-pvar1 (random!! 2)))
          (*when (<!! (self-address!!) length2)
             (*set bit-pvar2 (random!! 2)))

          ; Display the two binary numbers
          (pbp bit-pvar1 :length 20 :title "Bit-pvar 1")
          (pbp bit-pvar2 :length 20 :title "Bit-pvar 2")

          ; Display the result of adding them
          (pbp (very-long-add!! bit-pvar1 bit-pvar2)
             :length 20 :title "Result    "))
    (values)))
```

Here's the actual output of two sample calls to the **test-very-long-add!!** function:

```
> (test-very-long-add)
Bit-pvar 1: 0 0 0 0 0 0 1 0 1 1 1 0 1 0 0 1 0 1 0 1
Bit-pvar 2: 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 1 0 1 0
Result    : 0 0 0 0 0 0 1 1 1 1 1 0 0 1 0 1 1 1 1 1
> (test-very-long-add)
Bit-pvar 1: 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1 1 1 1 0 1
Bit-pvar 2: 0 0 0 0 0 1 0 0 0 1 0 0 1 0 0 0 1 0 1 0
Result    : 0 0 0 0 0 1 0 0 1 0 0 1 1 1 0 0 0 1 1 1
```

Obviously, we're only adding numbers of 12 to 17 bits in these examples, but **very-long-add!!** can be used without modification to add numbers with thousands or even millions of bits, limited only by the number of processors that you are currently using.

For the Curious: If you'd like to get a closer look at the inner workings of **very-long-add!!**, you can use the *Lisp function **ppp!!** to display the value of any pvar expression without affecting the value that expression returns. Unlike **ppp**, which prints a pvar and then returns **nil**, the **ppp!!** function prints a pvar and then returns the same pvar as its value. For more information, see the entry for **ppp!!** in the **Lisp Dictionary*.

6.2 Fit the Second: A Segmented news!! Function

In Chapter 3 we looked at router and grid communication, and I pointed out that grid communication is faster because of its regularity: all values move in the same direction across the grid. The **scan!!** function is also preferable to using the router, for the same reason: it's faster and more efficient. But scanning is much more flexible than grid communication. You can use **scan!!** to write data-movement operations that have the speed of grid communication operations, yet at the same time allow you to rearrange the values of your pvars in router-like ways.

For example, the grid communication operator **news!!** doesn't have a **:segment-pvar** argument like **scan!!**, but you can use segmented **scan!!** calls to perform **news!!**-like shifts of data in a segmented manner.

As a specific example, let's take the pvar returned by **(self-address!!)** as our "data" pvar:

```
> (ppp (self-address!!) :end 20 :format "~2D ")
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
```

and suppose that we have a segment pvar with the following values (for clarity, I've used the **:format** argument of **ppp** to replace **nil** values with dots):

```
> (ppp segment-pvar :end 20 :format " ~:[.~;T~] ")
 T  T  .  .  T  .  .  .  T  .  .  .  T  .  .  .  T  .  .  .
```

We can use a segmented **scan!!** to "rotate" each segment of the **self-address!!** pvar one position to the right, producing:

```
 0  3  1  2  7  4  5  6 11  8  9 10 15 12 13 14 19 16 17 18
```

Here's how to do it:

```
> (defun segmented-news!! (pvar segment-pvar)

    (*let  (end-segment-pvar
            result
            temp)

    ; Define a second segment pvar that has T's at
    ; the _end_ of the segments defined by segment-pvar
    (*set end-segment-pvar
      (scan!! segment-pvar 'copy!! :direction :backward
              :segment-pvar t!! :include-self nil))
```

```
; Last active processor in end-segment-pvar must be T
(*setf (pref end-segment-pvar (*max (self-address!!))) T)

; use scan!! to shift pvar values forward one position
(*set result
  (scan!! pvar 'copy!! :segment-pvar t!! :include-self nil))

; use a backward scan to copy the last value
; of each segment back to the start of the segment
(*set temp
  (scan!! pvar 'copy!! :segment-pvar end-segment-pvar
      :include-self t :direction :backward))

; combine the copied last elements from temp pvar
; with the elements in the result pvar
(*when segment-pvar (*set result temp))

; return the resulting shifted pvar
result))
```

There are three **scan!!** calls in this function, each of which performs a particular task. In order, the scans are:

- A "backward" **copy!!** scan with a segment pvar of **t!!** and **:include-self nil**. This shifts the contents of **segment-pvar** "backwards" by one element, producing a segment pvar that contains **t** for every processor that ends a segment. (Note that we also have to use ***setf** to make sure this pvar has a **t** in the "last" active processor.)

- A "forward" **copy!!** scan with a segment pvar of **t!!** and **:include-self nil**. This shifts each value of the supplied **pvar** argument "forward" by one step.

- A "backward" **copy!!** scan using the end-of-segment pvar we created, to copy the last value of each segment back to the first location in the segment. Notice that this time **:include-self** is **t**, to keep copied values from crossing segment boundaries.

The **result**, after we use ***when** and ***set** to combine things, is a copy of **pvar** with its contents rotated forward by one element within each segment, as defined by **segment-pvar**.

Here's an example of this function in action, using the example shown at the beginning of this section:

```
> (*defvar segment-pvar (zerop!! (mod!! (self-address!!) 4)))
SEGMENT-PVAR
> (*setf (pref segment-pvar 1) T) ; set processor 1 as well

> (ppp (self-address!!) :end 20 :format "~2D ")
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19

> (ppp segment-pvar :end 20 :format " ~:[.~;T~] ")
 T  T  .  .  T  .  .  .  T  .  .  .  T  .  .  .  T  .  .  .

> (ppp (segmented-news!! (self-address!!) segment-pvar)
       :end 20 :format "~2D ")
 0  3  1  2  7  4  5  6 11  8  9 10 15 12 13 14 19 16 17 18
```

For Perfectionists: As defined above, the **segmented-news!!** function calculates an **end-segment-pvar** every time it is called. Calling **segmented-news!!** repeatedly (for instance, within a loop) would be very inefficient. A better method is to rewrite this function to accept an **end-segment-pvar** argument, and then precalculate the end-segment pvar before calling **segmented-news!!**. And even if you're not using scans within a loop, precomputing the start-segment and end-segment pvars is a very practical method for using scans, because (as shown in the **segmented-news!!** example) it makes it easy for you to use both "forward" and "backward" scanning in your functions.

6.3 Fit the Third: Using the Router Efficiently

Router communication, while slower on average than grid communication, is nevertheless a very flexible way to move data between processors. There are many cases in which it might seem at first that using router communication is highly inefficient, yet by a suitable rearrangement of data even these extreme cases can be handled effectively.

As an example, consider the "many collisions" get operation. This happens when the address pvar argument of a **pref!!** call causes many processors to request a value from a single address, producing many "collisions" of requests. The router has built-in hardware for handling these collisions effectively, but when the number of requests is very large the hardware algorithm can run out of memory. In these cases, a slower software algorithm is used instead to perform the get operation.

By using a ***defstruct** pvar, a sort and a **scan!!** or two, we can simulate this algorithm, and see what steps the router has to take to handle a "many collisions" get.

The ***defstruct** pvar is used to hold information about the data transfers that we make:

```
> (*defstruct gmc-data
   (fetch-address nil :type fixnum)
   (originating-address nil :type fixnum))
```

For each processor, the **fetch-address** slot of this pvar records the address from which the processor is requesting a value, and the **originating-address** slot records the self address of the processor itself, so that the data can be sent back to it.

The function we write will need to do the following:

- Create a **gmc-data** pvar containing the **fetch-address** and **originating-address** for each active processor.
- Sort the values of this pvar by **fetch-address**, to gather together requests for data from the same **fetch-address**, and at the same time "pack" the requests together into low-address processors so that there are no gaps between them.
- Find the first processor in each group of similar requests, and do a "get" (**pref!!**) operation to retrieve values for just those processors.
- Do a **scan!!** to copy the retrieved values to all other processors in each group of requests.
- Finally, do a "send" (***pset**) operation to deliver the retrieved values to the processors that requested them.

The entire algorithm can be written as a single function:

```
> (defun pref!!-many-collisions (data address)
   (let ((number-of-active-processors (*sum 1)))
    (*let ((sort-data (make-gmc-data!!
                        :fetch-address address
                        :originating-address (self-address!!))))

      ; Sort sort-data pvar by fetch-address and
      ; pack into low-address processors so it is contiguous
      (*pset :no-collisions sort-data sort-data
         (rank!! (gmc-data-fetch-address!! sort-data) '<=!!))
```

```
; Select processors that contain data after the sort
(*all
 (*when  (<!!  (self-address!!)
               number-of-active-processors)

  ; Create segment pvar that is T for the first element
  ; of each group of elements with the same fetch-address
  (*let ((beginning-of-segment
          (or!! (zerop!! (self-address!!))
           (/=!! (gmc-data-fetch-address!! sort-data)
              (scan!! (gmc-data-fetch-address!! sort-data)
                      'max!! :include-self nil))))
         fetched-data)

  ; Do a get only for the first element of each segment
  (*when beginning-of-segment
    (*set fetched-data
     (pref!! data (gmc-data-fetch-address!! sort-data))))

  ; Use scan!! to copy fetched data to the other elements
  (*set fetched-data
    (scan!! fetched-data 'copy!!
       :segment-pvar beginning-of-segment))

  ; Use originating-address slots to send the data back
  ; to the processors that originally requested it
  (*pset :no-collisions fetched-data fetched-data
    (gmc-data-originating-address!! sort-data))

  ; Return the redistributed data
  fetched-data)))))
```

Here's an example of how you would call **pref!!-many-collisions**:

```
> (*defvar collision-addresses (floor!! (self-address!!) 6))
COLLISION-ADDRESSES

> (ppp collision-addresses :end 28)
0 0 0 0 0 0 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3 4 4 4 4

> (ppp (pref!!-many-collisions (*!! (self-address!!) 3)
                               collision-addresses)
       :end 28)
0 0 0 0 0 0 3 3 3 3 3 3 6 6 6 6 6 6 9 9 9 9 9 9 12 12 12 12
```

There are two **scan!!** calls in **pref!!-many-collisions**, each having a different set of arguments:

- The first **scan!!** call does a forward **max!!** scan of the sorted **sort-data** values, with **:include-self nil**. This causes each processor to compare its own value with the maximum value of all the preceding processors. Only processors that start a segment of similarly addressed requests will see a difference between these values, so this step quickly locates the first processor in each segment.
- The second **scan!!** call is a forward **copy!!** scan using the segment pvar derived from the first **scan!!**. The first processor in each segment has retrieved a value, so this **scan!!** operation copies that value to the other processors of the segment.

Notice how in this example we're using one type of **scan!!** call to determine the bounds for another type of **scan!!** call. It's the multi-faceted nature of **scan!!** that makes this possible.

Notice also the combination of ***pset** and **rank!!** at the beginning of the **pref!!-many-collisions** function. This is an important example of a pair of *Lisp functions working in tandem to perform more than one operation. The **rank!!** function provides a ranking of the **fetch-address** values in the **sort-data** pvar, and ***pset** uses this ranking to sort the values in ascending order. The important point to notice, however, is that some processors might be deselected when **pref!!-many-collisions** function is called. When **rank!!** does its ranking, these processors are simply ignored. The result is that *pset "packs" all the active values of **sort-data** into processors with low addresses, eliminating any "gaps" caused by deselected processors.

For The Curious: This "packing" of pvar values is a useful trick for eliminating gaps in your data set. We can rewrite the ***pset-rank!!** combination used in **pref!!-many-collisions** to "pack" the values of a pvar without changing their order, as in:

```
(*pset :no-collisions pvar pvar
                      (rank!! (self-address!!) '<=!!))
```

The expression **(rank!! (self-address!!) '<=!!)** is an important enough idiom that there is a *Lisp function that does the same thing: **enumerate!!**. So this "pack" operation can be written more efficiently as:

```
(*pset :no-collisions pvar pvar (enumerate!!))
```

And, by the way, it shouldn't suprise you to learn that **scan!!** can be used to write a simple version of **enumerate!!**:

```
(defun enumerate!! () (scan!! 1 '+!! :include-self nil))
```

6.4 Fit the Fourth: Creating a "List" Pvar

While *Lisp does not have a direct parallel analogue of the list data structure, you can use *Lisp operations to define a parallel data structure that has much of the flexibility of lists. This example uses VP sets, router communication, and scanning to define a parallel data structure with an arbitrary number of elements per processor, which I call a "list" pvar on account of its flexible nature. (Lisp purists will most likely prefer some other name.)

This is accomplished by defining a new VP set that has sufficient processors to hold all the elements of the list pvar, and by defining the list pvar in such a way that it is divided into segments of elements, one segment for each processor in the original VP set. In this way, each processor in the original VP set is assigned a "list" of elements for its use, of any required length.

Note: This section introduces some *Lisp functions and variables that we haven't looked at in previous sections. In particular, this section introduces:

> ***minimum-size-for-vp-set***
> **allocate-processors-for-vp-set**
> **deallocate-processors-for-vp-set**
> **next-power-of-two->=**

There are definitions and examples of each of these operations in the **Lisp Dictionary*.

Here's the definition of the VP set:

```
> (def-vp-set list-vp-set nil
    :*defvars ((segment-pvar nil) (elements nil)))
```

It is defined as a *flexible VP set*, meaning that its size and shape are determined at run time. It has two associated pvars, an **elements** pvar that will contains the elements of the lists we define, and a **segment-pvar** (as used in the **scan!!** examples above) that will indicate with a **t** value the beginning of each segment of list elements.

We'll also want a pair of permanent pvars that describe the start location and length of the segment of elements assigned to each processor:

```
> (*defvar *list-start-processor*)
> (*defvar *number-of-elements*)
```

Now, here is the macro that actually defines and allocates the **list-vp-set** for the duration of a body of *Lisp code:

```
> (defmacro allocate-list-pvar (length value &body body)
   `(let ((total-processors-required (*sum ,length)))

      ;;; Allocate processors in list-vp-set for list elements
      (allocate-processors-for-vp-set list-vp-set
         (list (max *minimum-size-for-vp-set*
            (next-power-of-two->= total-processors-required))))
      (*with-vp-set list-vp-set
         (*set segment-pvar nil elements nil))

      ;;; Get send addresses of elements in that list-vp-set
      ;;; that will contain the first element of each segment
      (*let ((*list-start-processor*
              (scan!! ,length '+!! :include-self nil))
         (*number-of-elements* ,length))

      ;;; For processors that have requested a non-zero number
      ;;; of elements, send the initial values to the first
      ;;; element of the corresponding segments.
      (*when (plusp!! ,length)
         (*pset :no-collisions ,value elements
            *list-start-processor* :notify segment-pvar))

      ;;; Use scan!! to copy the initial value to all elements
      ;;; in each segment.
      (*with-vp-set list-vp-set
       (*set elements
        (scan!! elements 'copy!! :segment-pvar segment-pvar)))

      ;;; Evaluate body forms with list-vp-set defined
      (progn ,@body )

      ;;; Deallocate the list-vp-set so that it can be reused.
      (deallocate-processors-for-vp-set list-vp-set))))
```

Notice that the **allocate-list-pvar** macro simply defines the new VP set, but does not select it; the **body** of code enclosed by the macro is free to select the **list-vp-set** whenever it is needed.

As an example of what you can do with a "list" pvar once you've defined it, here's a function that prints out the first *n* lists in the **list–vp–set**:

```
> (defun print-lists (n)
    (dotimes (i n)
      (format t "~&( ")
      (let  ((start-address (pref *list-start-processor* i))
             (length (pref *number-of-elements* i)))
       (*with-vp-set list-vp-set
        (dotimes (i length)
         (format t "~D "
            (pref elements (+ start-address i))))))
      (format t ")")))
```

Finally, here's a test function that allocates the **list–vp–set** with a random set of list lengths, calls **ppp** to display the first *n* lengths it assigned, and then calls **print–lists** to display the first *n* lists in the **list–vp–set**:

```
> (defun test-lists (n)
   (*let ((lengths (random!! 8)))
      (ppp lengths :end n)
      (allocate-list-pvar lengths (self-address!!)
         (print-lists n))))
```

Here's what the output from **test–lists** looks like:

```
> (test-lists 6)
4 3 5 4 3 2
( 0 0 0 0 )
( 1 1 1 )
( 2 2 2 2 2 )
( 3 3 3 3 )
( 4 4 4 )
( 5 5 )

> (test-lists 6)
2 5 5 1 0 4
( 0 0 )
( 1 1 1 1 1 )
( 2 2 2 2 2 )
( 3 )
( )
( 5 5 5 5 )
```

While it's not a simple matter to dynamically change the size of the list assigned to each processor, the "list" pvar has many of the advantages of the list data structure in Common Lisp.

There are many applications where this ability to assign a "list" of elements to each original processor comes in handy. For example, in simulating the behavior of subatomic particles, a list pvar can be used to keep track of the particles that result from collisions or decay.

Another example is in parallel graphics programs, where a number of polygons must be rendered (displayed on a screen). Each polygon can be broken down into triangles that are easy to render, but each polygon may produce a different number of triangles. A list pvar can be used to hold the set of triangles derived from each polygon, and then the entire list pvar can be passed to a simple triangle-rendering algorithm for display.

6.5 Summary: There's More Than One Way to Scan a Pvar

In this chapter, we've seen that:

- You can use **scan!!** to add extremely large integers.
- You can use the segmented version of **scan!!** to define other segmented operations.
- You can use **scan!!** along with ***defstruct** pvars, ranking tools, and communication operators to increase the efficiency of your data transfer operations.
- You can use **scan!!** along with the existing parallel data structures of *Lisp to define new data structures that have the kind of flexibility that you need.

In short, there are many interesting and unusual things you can do with *Lisp: the examples in this chapter are but a representative sample. The best way to learn about these unusual tricks is by creating them yourself. Decide what kind of operation you wish to perform, and then combine *Lisp operators to implement it.

"Fish say, they have their stream and pond;
But is there anything beyond?"

Rupert Brooke

Chapter 7

Going Further with *Lisp

7.1 Topics We Haven't Covered

There are some *Lisp topics that we haven't covered in this Getting Started guide. You can find out more about them by looking in the **Lisp Dictionary*. They are described briefly below.

Creating and Using Array Pvars

*Lisp has parallel equivalents of most of the array and vector operations of Common Lisp. In this document, we've only seen one of these, **make-array!!**. Section 1.5 of the overview chapter in the **Lisp Dictionary* provides a complete list of all array pvar operations in *Lisp. Along with operations for array pvars, *Lisp includes specialized operations for vector pvars and bit-array pvars, and parallel equivalents of Common Lisp sequence operators, for example **find!!**, **length!!**, **position!!**, and **substitute!!**. These are also listed in Section 1.5.

Turning Array Pvars "Sideways"

If you use array pvars heavily in your *Lisp code, you may want to consider turning your arrays "sideways." This operation changes the arrangement of data in your array pvars so that the CM can access them more efficiently. For more information about "sideways" arrays, see the dictionary entries for **sideways-aref!!** and ***sideways-array**.

Creating and Using *defstruct Pvars

In this guide, we've barely touched on the use of ***defstruct** for defining structure pvars, but structure pvars are a powerful tool for defining your own parallel data types. For more information about structure pvars and examples of how to create them, see the dictionary entry for ***defstruct**.

Dynamically Allocating Blocks of CM Memory

Along with permanent, local, and temporary pvars, there is a fourth kind of pvar known as a *global* pvar. Allocated by the *Lisp operator **allocate!!**, global pvars are used to allocate and reference large amounts of CM memory within your *Lisp programs. Whereas you will typically allocate permanent pvars one at a time, you can allocate dozens or even hundreds of global pvars at one time, and store them in a list to be used whenever you need them. For more information, see the dictionary entry for **allocate!!**.

Defining Segment Set Objects for Scanning

Along with the **scan!!** operator, there is a more advanced scanning function, **segment-set-scan!!**, that uses a "segment set" data structure to define where the segments of a pvar begin and end. These segment set data objects are created by the function **create-segment-set!!**, and there are a large number of *Lisp functions for accessing the contents of these segment set objects. For more information, see the dictionary entries for **segment-set-scan!!** and **create-segment-set!!**.

Just For Fun: Controlling The Front Panel LEDs

Finally, *Lisp includes a function called ***light** which takes a boolean pvar and calls an internal Paris function to set the state of the front panel LEDs of the CM. For more information, see the dictionary entry for ***light**, and also the entries for the Paris operators **cm:latch-leds** and **cm:set-system-leds-mode** in the *Connection Machine Parallel Instruction Set (Paris)* manual.

7.2 Where Do You Go from Here?

I hope this Getting Started guide has given you the start you need towards proficient *Lisp programming. Here are some suggested sources to which you can turn for further information about *Lisp programming and more examples of *Lisp code:

- The **Lisp Dictionary* contains a complete list of all *Lisp functions and macros, as well as information on type declarations and compiler options that you'll find very handy.
- The directory **/cm/starlisp/interpreter/f6100** contains a number of example *Lisp files. The names of these files are listed in the file **examples–def–file–set.lisp** in this directory.
- Your fellow *Lisp programmers are a good source of information, sample code, and help in debugging recalcitrant programs. Ask around among the *Lisp people you know for help and advice.
- Finally, you can contact Thinking Machines Corporation Customer Support for help and advice on *Lisp programming, via the email address given in the front of this guide.

May all your parentheses balance, and may all your CM processors be active!

Appendixes

Appendix A

Sample *Lisp Application

This chapter contains commented source code for the cellular automata example used in Chapter 1, along with extensions that make the system more generic. This file is also available on-line in the *Lisp software directory, in the file

```
/cm/starlisp/interpreter/f6100/cellular-automata-example.lisp
```

Check with your system administrator or applications engineer if you need help locating this file.

A.1 Cellular Automata Example

```
;;; CA Example From "Instant *Lisp" Chapter
;;; by William R. Swanson, Thinking Machines Corporation


;;; Load into *Lisp package
(in-package '*lisp)


;;; --- Global Variables ---
;;; This defines a permanent pvar to hold the grid of cells
(*defvar *automata-grid* 0)

;;; Total number of states allowed per cell
(defvar *total-number-of-states* 10)

;;; Cell "neighborhood" to use for automata
(defvar *neighborhood* :neumann)
```

```
;;; --- Simple Tools ---

;;; Function to display the grid
(defun view (&optional (width 8) (height 5))
  (ppp *automata-grid* :mode :grid :end (list width height)))

;;; Functions to read/write individual cells
(defun read-cell (x y)
  (pref *automata-grid* (grid x y)))

(defun set-cell (x y newvalue)
  (*setf (pref *automata-grid* (grid x y)) newvalue))

;;; Function to set value of entire grid
(defun set-grid (newvalue)
  (*set *automata-grid*
     (mod!! newvalue *total-number-of-states*)))

;;; Function to randomly set the value of each cell
(defun random-grid ()
  (set-grid (random!! *total-number-of-states*)))

;;; Tools to set up a fixed pattern:
(defun set-cells (cell-list value)
  (dolist (cell cell-list)
    (set-cell (car cell) (cadr cell) value)))

;;; Clear grid, set up "r-pentomino" pattern, and display
(defun init ()
  (set-grid 0)
  (set-cells '((2 2) (3 1) (3 2) (3 3) (4 1))
          1)
  (view))


;;; Tools to get information about neighboring cells.
;;; Count non-zero Von Neumann neighbors
(defun neumann-count (grid)
  (+!! (news!! grid 0  -1)    ;; north
       (news!! grid 0  1)     ;; south
       (news!! grid -1 0)     ;; west
       (news!! grid 1  0)     ;; east
       )))
```

```
;;; Count non-zero Moore neighbors
(defun moore-count (grid)
    (+!! (news!! grid 0  -1) ;; north
      (news!! grid 0  1)  ;; south
      (news!! grid -1 0)  ;; west
      (news!! grid 1  0)  ;; east
      (news!! grid -1 -1) ;; northwest
      (news!! grid -1 1)  ;; southwest
      (news!! grid 1  -1) ;; northeast
      (news!! grid 1  1)  ;; southeast
      )))

;;; Count neighbors for current *neighborhood*
(defun neighbor-count ()
  (*let ((grid (signum!! *automata-grid*)))
    (ecase *neighborhood*
      (:moore (moore-count grid))
      (:neumann (neumann-count grid)))))


;;; Function to run the automata defined by the function one-step.
(defun run (&optional (n 1))
  (dotimes (i n)
    (set-grid (one-step))))

;;; Function to run automata for n steps and display the grid.
(defun view-step (&optional (n 1))
  (run n)
  (view))

;;; Tool to check whether all the cells are dead (zero).
(defun deadp ()
  (zerop (*sum (signum!! *automata-grid*))))


;;; --- Simple Automaton Example ---
;;; This automaton obeys the following rules:
;;; If a cell is:
;;;    EVEN - divide its value by 2
;;;    ODD  - add 1 to its value and multiply by 2

(defun one-step ()
  (if!! (evenp!! *automata-grid*)
     (floor!! *automata-grid* 2)
     (*!! (1+!! *automata-grid*) 2)))
```

```
;;; --- "9 Life" automata, based on Conway's Game of Life ---
;;; Obeys the following rules:
;;;  For each cell, count the number of non-zero neighbors.
;;;    If it is <1, or >3, subtract 1 (zero cells remain zero).
;;;    If it is 2 or 3, add 1
;;;    Otherwise, do nothing

(defun one-step ()
  (*let ((count (neighbor-count)))
    (cond!!
      ;; When count is <1 or >3, subtract 1 if not zero.
      ((or!! (<!! count 1) (>!! count 3))
       (if!! (zerop!! *automata-grid*)
          *automata-grid*
          (1-!! *automata-grid*)))

      ;; When count is 2 or 3, add 1
      ((<=!! 2 count 3) (1+!! *automata-grid*))

      ;; Otherwise, leave cells unchanged
      (t *automata-grid*))))

;;; --- Extension of Material in Chapter 1 ---
;;; Tools to define and run generic automata:

;;; Macro that defines a named automaton
;;; as a list of two function objects.
;;; Init function sets up the *automata-grid*
;;; Step function calculates and returns single "step" of automata

(defmacro defautomaton (name &key init step)
  `(progn
     (defvar ,name)
     (setq ,name (list '(lambda ,@init)
                       '(lambda ,@step)))
     ',name))

(defun init-function (automaton) (car automaton))
(defun step-function (automaton) (cadr automaton))
```

```
;;; Definitions for the two automata we've already written:
(defautomaton four2one
    ;;; init function takes no arguments, and randomizes the grid.
    :init ((&rest ignore)
        (setq *total-number-of-states* 10)
        (random-grid))
    ;;; step function takes no arguments, and calculates one step
    :step ((&rest ignore)
        (if!! (evenp!! *automata-grid*)
           (floor!! *automata-grid* 2)
           (*!! (1+!! *automata-grid*) 2))))

(defautomaton 9life
    ;;; init function takes optional arguments defining
    ;;; the current neighborhood, the initial pattern, and
    ;;; the value stored into cells that are part of the pattern
    :init ((&optional (neighborhood *neighborhood*)
                (start-pattern '((2 2) (3 1) (3 2) (3 3) (4 1)))
                (start-value 1))
        (setq *neighborhood* neighborhood
           *total-number-of-states* 10)
        (set-grid 0)
        (set-cells start-pattern start-value))
    ;;; step function takes no arguments, and
    ;;; calculates a single step of the automaton
    :step ((&rest ignore)
        (*let ((count (neighbor-count)))
          (cond!! ((or!! (<!! count 1) (>!! count 3))
                (if!! (zerop!! *automata-grid*)
                   *automata-grid*
                   (1-!! *automata-grid*)))
               ((<=!! 2 count 3) (1+!! *automata-grid*))
               (t *automata-grid*)))))

;;; --- Tools used to select an automaton to run ---

;;; Currently selected automaton
(defvar *current-automaton*)

;;; Function to select an automaton and initialize the grid
(defun setup (automaton &rest init-args)
  (setq *current-automaton* automaton)
  (initialize init-args))
```

```
;;; Function to call the init function of the current automaton,
;;; and display the initial state of the grid.
(defun initialize (&optional init-args)
  (apply (init-function *current-automaton*) init-args)
  (view))


;;; Function to run the automaton for n steps and display the grid
(defun run (&optional (n 1) &rest step-args)
  (dotimes (i n)
    (set-grid
      (apply (step-function *current-automaton*)
          step-args)))
  (view))


;;; The following sample session shows how to set up and
;;; run the automata defined by the above extensions:
;
;> (setup four2one) ;; Simple 4-2-1 loop automata
;
;5 7 9 3 2 3 1 9
;7 0 3 4 3 3 0 3
;9 4 2 3 8 9 2 5
;7 3 3 5 8 2 9 3
;1 7 9 1 8 6 9 6
;
;> (run)
;2 6 0 8 1 8 4 0
;6 0 8 2 8 8 0 8
;0 2 1 8 4 0 1 2
;6 8 8 2 4 1 0 8
;4 6 0 4 4 3 0 3
;
;> (run 50)
;4 1 0 2 2 2 1 0
;1 0 2 4 2 2 0 2
;0 4 2 2 1 0 2 4
;1 2 2 4 1 2 0 2
;1 1 0 1 1 4 0 4
;
```

```
;> (setup 9life :neumann) ;; 9 Life, Neumann neighborhood
;
;0 0 0 0 0 0 0 0
;0 0 0 1 1 0 0 0
;0 0 1 1 0 0 0 0
;0 0 0 1 0 0 0 0
;0 0 0 0 0 0 0 0
;
;> (run)
;0 0 0 0 0 0 0 0
;0 0 1 2 1 0 0 0
;0 0 1 2 1 0 0 0
;0 0 1 1 0 0 0 0
;0 0 0 0 0 0 0 0
;
;> (run 50)
;0 0 0 0 0 0 0 0
;0 0 0 8 3 0 0 0
;0 0 5 0 7 0 0 0
;0 0 4 5 9 0 0 0
;0 0 0 0 0 0 0 0
;
;> (setup 9life :moore) ;; 9 Life, Moore neighborhood
;
;0 0 0 0 0 0 0 0
;0 0 0 1 1 0 0 0
;0 0 1 1 0 0 0 0
;0 0 0 1 0 0 0 0
;0 0 0 0 0 0 0 0
;
;> (run)
;0 0 0 1 1 0 0 0
;0 0 1 2 2 0 0 0
;0 0 2 0 0 0 0 0
;0 0 1 2 1 0 0 0
;0 0 0 0 0 0 0 0
;
;> (run 50)
;0 0 0 0 4 1 0 1
;5 7 7 2 1 1 1 0
;0 0 0 0 1 1 5 2
;4 4 9 2 1 0 4 0
;0 0 0 0 1 2 0 2
```

Appendix B

A *Lisp/CM Primer

For those readers unfamiliar with the Connection Machine (CM) system, this chapter provides a brief overview of data parallel processing and the CM, and also describes how the *Lisp language fits into the CM system.

B.1 Data Parallel Processing

Most computers are designed according to a *serial processing* model in which a single computing unit, or *processor*, executes a single program one instruction at a time. The Connection Machine system differs from such serial computing systems in two ways: the CM is a *parallel processing* system, and on top of that the CM is a *data parallel* processing system.

Parallel processing means having more than one processor executing your program at the same time. The CM is a *massively parallel* processing system; it applies many thousands of processors to the execution of your program simultaneously. The main difficulty with executing programs in parallel lies in coordination: getting all those thousands of processors to work together efficiently in solving problems for you. The CM solves this problem through *data parallel processing*.

In data parallel processing a large number of processors, each with its own associated portion of memory, executes identical computing operations simultaneously. That is, each processor performs the same identical operation on its own portion of memory. This avoids the problem of coordinating the actions of all those processors; all processors do the same thing at the same time. But while all processors perform the same operation, each processor does so on a different piece of data. This is the key point of data parallel processing: it's like executing the same program simultaneously on many thousands of pieces of data.

Here are some examples of how data parallel processing can be used:

- A text-retrieval program might store a set of articles one-per-processor and then have each processor search its particular article for a key word or phrase.
- A graphics program might arrange the pixels of an image one-per-processor and then have each processor calculate the color value of its pixel, all at the same time.
- A finite-state automata program (for example, Conway's Game of Life) might create a large number of cells stored one-per-processor. Each cell would have a small number of possible states, and all the cells would be simultaneously updated at each "tick" of a clock according to a set of fixed rules.

Programs written for data parallel processing systems have a unique style, and the process of developing programs for these types of machines is called *data parallel programming*.

B.2 Data Parallel Processing on the CM

The model of data parallel processing used on the CM is as follows:

A single processor, called the *front end*, executes a program in either a high-level language like *Lisp or in a low-level parallel programming language like Paris. As the program runs, the front end transmits instructions and data to the CM (see Figure 10).

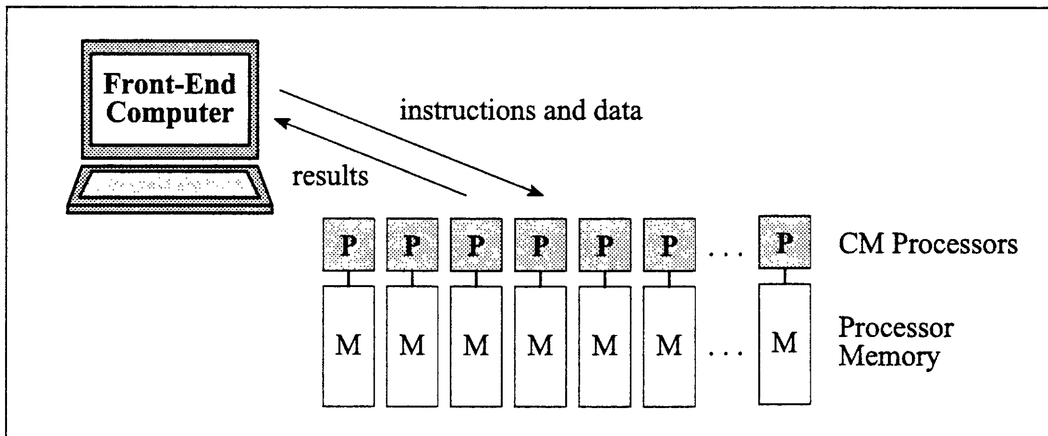


Figure 10. The data parallel programming model of the CM

The processors within the CM store the data they receive in their own areas of memory, and execute the instructions they receive simultaneously. The CM processors can also transmit the results of their computations back to the front end.

In the Connection Machine system, the front end is a standard serial processing computer such as a Sun Microsystems Sun-4, a Digital Equipment Corporation VAX computer, or a Symbolics 3600 series Lisp machine.

As the front end executes its program, serial operations (those that don't require parallel computation) are executed directly by the front end. Whenever the front end comes to an operation that requires parallel computation, it transmits that instruction to the CM, to be executed by the CM processors.

Thus, serial code within a program is executed on the front end computer in the usual manner; parallel code is executed by the CM processors. The front end and CM operate independently; the front end can be carrying out a serial computation (say adding up your restaurant bill) while the CM carries out a parallel computation (such as working out what the same order would cost you in every restaurant in the country).

For the Curious: How It's Really Connected

The picture presented in Figure 10 above is really a simplification. The actual physical connection between the front end and the CM is more involved. (See Figure 11.)

The front end communicates with the CM via a Front End Bus Interface (FEBI) card. There can be one or more of these cards in the front end; each provides a separate link to a CM. Each FEBI is connected via an H-bus cable to a central switching board, the Nexus, inside a CM. The Nexus acts as a manager for the resources of the CM, allowing a front end to request that some, all, or none of the CM processors be made available for its use.

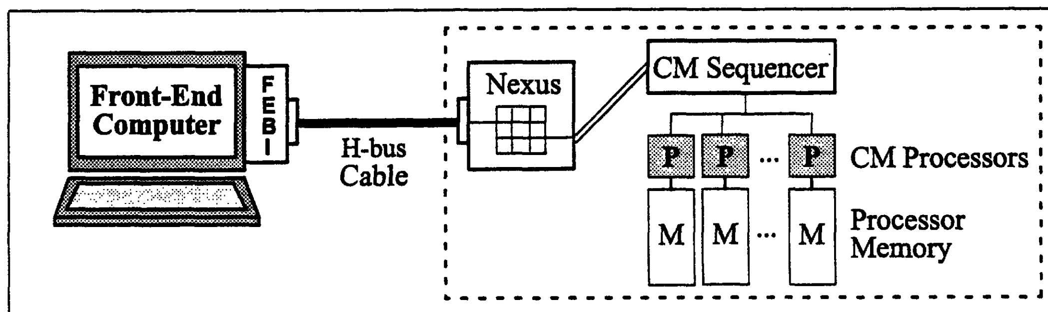


Figure 11. The real connection between the front end and the CM

Processors within the CM are grouped in *sections*, each of which is managed by a single *sequencer*. It is the sequencer that handles the translation of instructions received from the front end into instructions that the individual processors can execute. Depending on the hardware you have available, there may be one, two, or four sequencers in the CM, each with its own associated group of processors.

The Nexus makes processors available to the front end by grouping sequencers together. Thus, if your CM has four sequencers, your front end may be connected to one, two, or all four of them, depending on the state of the Nexus. This is all transparent to a user on the front end, of course. Regardless of the number of sequencers that are in use, it always appears as if there is just one computational entity waiting for instructions at the end of that long, black H-bus cable.

It's not essential that you know all this to program the CM; for most purposes the picture of a single front end attached directly to a single CM is sufficient. However, keeping the more complex picture of front-end/CM relations in the back of your mind can help you in understanding the commands you use and the warning messages you see when you execute your program on a CM.

B.3 Other Features of the CM System

Processor Selection

A program can specify that only a particular set of CM processors should carry out a sequence of operations. In the text retrieval example mentioned above, the processors that find a particular word or phrase in their articles might be instructed to search further for another word or phrase, while the remaining processors are idle. The set of processors that is currently selected to perform CM operations is commonly referred to as the *currently selected set* of processors, or often just as the *active* processors.

Processor Communication

Processors can pass messages to each other. The processors in the CM are connected by a general message-passing network called the *router*. The router allows each processor to send a piece of data to any other processor in the machine, with all processors transmitting data simultaneously, in a process known as *router communication*. In addition, the CM system has a faster form of communication called *grid communication*, which in certain cases allows processors to pass values to each other more efficiently.

Virtual Processors

Different CM models have different numbers of processors available for use by programs. The hardware of a CM always has some fixed number of *physical* processors; this may be 4K, 8K, 16K, 32K, or 64K processors, but there is always some magic number that represents the actual number of processors that are "really there" inside the CM.

This does not limit the size of the data set that a program can use, however. If more processors are required than are available in the hardware of the CM, each physical processor simulates the actions of two or more *virtual* processors by dividing up its memory accordingly and performing each operation multiple times.

Because of this, the number of virtual processors must always be a power-of-two multiple of the number of physical processors that are available within the CM. Thus a CM with 16K physical processors can operate as if it has 32K processors, 64K processors, etc.

A single CM can even simulate different numbers of virtual processors at the same time. It is possible to define a number of *virtual processor sets* (VP sets), each having a different number of virtual processors, and to use them together in the same program to manipulate data sets of different sizes.

B.4 The *Lisp Language

Given that the Connection Machine system consists of two machines, a front end and the CM itself, where does *Lisp itself fit into the picture?

B.4.1 A Front End Language with Side Effects on the CM

The *Lisp language is an extension of the Common Lisp standard; *Lisp programs run on the front end just like any other Common Lisp program. However, *Lisp provides programming constructs and data structures that are used to control the operations of the CM. The *Lisp language can therefore be thought of as a front-end language with side effects that cause computations to take place on the CM.

Because *Lisp is an extension of Common Lisp, *Lisp programs must be run from within a Common Lisp environment on the front end. For Sun front ends this is Sun Common Lisp. For Digital Equipment Corporation VAX computers, Lucid Common Lisp is used. On Symbolics 3600-series Lisp machines, *Lisp runs on top of Genera. No matter where you develop and run your *Lisp code, however, you always have available to you the tools and features of the Lisp programming environment provided by your front end.

B.4.2 Many Options for Executing Your Code

*Lisp comes in many forms, offering you many different ways to execute your code. You can run your *Lisp code "live" on physical CM hardware, in either interpreted or compiled form. You can also run your *Lisp code "simulated" on the *Lisp simulator, when CM hardware is not available. You can run your *Lisp code in batch, and under timesharing. This section describes each of these options in more detail.

The *Lisp Interpreter and Compiler

As with all Common Lisp systems, *Lisp has an interpreter and a compiler. The *Lisp interpreter is simply the standard interpreter of your Common Lisp environment, extended to understand and execute *Lisp operations. Each interpreted *Lisp operation makes one or more calls to Paris, the system software of the Connection Machine.

The *Lisp compiler is likewise an extension of the Common Lisp compiler, and is invoked whenever you compile a function or region of code. The *Lisp compiler translates *Lisp code into highly efficient Lisp/Paris code.

The *Lisp Simulator

Obviously, you can only run your *Lisp code "live" on CM hardware when there are processors available for you to use. Depending on the demand for the CM at your site, this could mean either a long wait or working in the wee hours of the morning.

For this reason, there is also a *Lisp simulator. This is a Common Lisp program that runs entirely on the front-end machine, simulating via software the operations of a permanently attached CM. The simulator allows *Lisp code to be tested and debugged when Connection Machine hardware is not available. The *Lisp simulator is freely available for the asking from Thinking Machines Corporation Customer Support.

The *Lisp simulator can be run at *any* time, regardless of whether hardware is available. The simulator is designed to be compatible with the hardware version of *Lisp; programs may be ported from one to the other with little or no modification. Also, the simulator is written in portable Common Lisp, so it can be used on *any* computer with a Common Lisp environment—CM hardware is *not* required to use the simulator.

Running *Lisp in Batch

*Lisp code can be executed in batch mode on the CM. Doing this requires that you add extra code to make your program execute properly when run under batch processing. The *CM User's Guide* contains more information, along with examples of running *Lisp code in batch.

Running *Lisp under Timesharing

*Lisp will also run on a CM that has timesharing enabled. To use *Lisp on a timeshared CM, you must use a version of the *Lisp software that includes special timesharing support code. Check with your systems administrator or applications engineer for more information on running *Lisp under timesharing.

Appendix C

All the Paris You Need to Know

The Connection Machine Parallel Instruction Set, affectionately nicknamed "Paris," is the low-level parallel programming language of the CM. In using *Lisp on a CM there are a few essential Paris operations with which you'll need to be familiar. This appendix provides a brief description of each of them. For more information about these operations, see Chapter 7, "In the Lisp Environment," of the *CM System User's Guide*.

For Simulator Users: The operations described here are all part of the Paris substrate of *Lisp, so you won't be able to call them when you're using the *Lisp simulator.

C.1 Attaching to a CM: (cm:attach)

The Paris function **cm:attach** attaches you to a CM so that you can start sending it commands. You can call it without any arguments, as in

```
> (cm:attach)
```

to attach to whatever CM hardware is available, or you can supply arguments that select a specific CM size, front-end interface, and even a specific section of a CM. You can even instruct Paris to wait (via the **:wait-p** argument) until the CM hardware is available.

```
> (cm:attach :8kp :interface 0) ;; Attach to 8K CM on interface
> (cm:attach :ucc0 :interface 0 ;; Attach to section 0 of CM
             :wait-p t)         ;; Wait until CM is available
```

Note: You normally don't need to call **cm:attach**. The *Lisp function ***cold-boot** will automatically call **cm:attach** for you if you are not currently attached to a CM. If you do call **cm:attach** yourself, remember that you must also call ***cold-boot** to initialize *Lisp.

C.2 Finding Out What CM Resources Are Available

The Paris **cm:finger** function is used to find out what CM hardware is available at your site. For example, a typical call to **cm:finger** might look like:

```
> (cm:finger)
CM        Seqs Size Front-end I/F  User      Idle      Command
----------------------------------------------------------------
LANA      0    4K   cyclops   2    Zener     00:23     starlisp
LANA      ---  ---  cyclops   0    (nobody)
     256K memory, 32-bit floating point
     1 free seqs on LANA -- 1 -- totalling 4K procs
```

This shows that the user "Zener" is attached to sequencer 0 of the CM named "LANA," and is using *Lisp. It also shows that sequencer 1 of LANA is available, and that you can attach to it via interface 0 from the front-end machine "cyclops." You can do this by typing

```
> (cm:attach :4kp :interface 0)
```

or simply by typing

```
> (cm:attach)
```

C.3 Finding Out if a CM is Attached: (cm:attached)

The Paris function **cm:attached** is a boolean test that checks whether a CM is currently attached. It returns two values. The first value is **t** when a CM is currently attached and **nil** otherwise. The second value is always **nil** unless an error occurred while looking for a CM.

For example:

```
> (cm:attached)     ;; With a CM attached, returns...
T
NIL
```

C.4 Timing *Lisp Code

The Paris function **cm:time** is used to time a section of code. You can wrap a call to **cm:time** around any section of *Lisp code to see how long it takes to execute. The **cm:time** function displays three kinds of information: the total elapsed time taken by your code, the amount of time the CM was actually active executing your code, and the percentage of the total elapsed time that the CM was active.

For example:

```
> (cm:time (sort!! (random!! 10) '<=!!))

Evaluation of (SORT!! (RANDOM!! 10) (QUOTE <=!!)) took 0.016738
seconds of elapsed time,
during which the CM was active for 0.011261 seconds
or 67.00% of the total elapsed time.
```

In this example, the total elapsed time was approximately 17 milliseconds, the CM active time was about 11 milliseconds, and the percentage of CM active time versus elapsed time was 67%.

C.5 Detaching from a CM: (cm:detach)

The Paris function **cm:detach** is the opposite of **cm:attach**, and is used to detach the CM so that other users can attach to it. It is called without any arguments, as in

```
> (cm:detach)
```

Note: Simply exiting from *Lisp will, in most cases, detach you from any CM to which you were attached. However, it is considered good form to call **cm:detach** yourself so that you don't accidentally leave the CM tied up when you exit.

Appendix D

Sample *Lisp Startup Sessions

The following are annotated examples of *Lisp hardware and simulator sessions on a Sun-4 front end, showing you the output that you're likely to see in starting up either version of *Lisp. All annotations are included as comments, of the form #|...|#. (**Note:** These comments are *not* displayed by *Lisp itself.) Commands that you type are preceded either by the UNIX "%" prompt or the Sun Common Lisp ">" prompt.

D.1 *Lisp Hardware Startup Session

```
#|   To start up *Lisp, you must type the UNIX-level command that loads
     the *Lisp software. In this sample session, it is: |#

% /usr/local/starlisp

#|   The first messages that you see are printed by Sun Common Lisp,
     showing the version of Common Lisp you are using,
     along with various copyright notices.  |#

;;; Sun Common Lisp, Development Environment 3.0.5 (Rev 01), 30-Aug-90
;;; Sun-4 Version for SunOS 4.0
;;;
;;; Copyright (c) 1985, 1986, 1987, 1988 by Sun Microsystems, Inc., All
Rights Reserved
;;; Copyright (c) 1985, 1986, 1987, 1988 by Lucid, Inc., All Rights Re-
served
;;;
;;; This software product contains confidential and trade secret
;;; information belonging to Sun Microsystems.  It may not be copied
;;; for any reason other than for archival and backup purposes.
```

```
#|  Following this are messages printed by Thinking Machines' Corporation
    software. |#

#|  The first thing the *Lisp software does is look for the
    "CM initializations" file. This file contains commands to perform any
    required initialization steps, such as loading software patches. |#

;;; Loading source file "/cm/patch/initializations.lisp"

#|  Depending on the version of *Lisp you are using, you may or may not
    see the next message, which informs you of the current state of
    the Lucid Common Lisp compiler: |#

;;; You are using the compiler in production mode (compilation-speed = 0)
;;; Generation of argument count checking code is enabled (safety = 1)
;;; Optimization of tail calls is enabled (speed = 3)

#|  One of the last things the "CM initializations" file does is print a
    message telling you the current patch level of *Lisp (that is, the
    total number of patch files that have been loaded). If you need to
    report a problem to Thinking Machines Corporation Customer Support,
    you should include the current patch level of your software, so
    that Customer Support personnel will know which patches you have
    loaded. In this case, 26 patches have been loaded, so *Lisp is at
    "Patch Level 26". |#

;;; *Lisp Patch Level 26

#|  Next, a header is printed showing the current version of the Thinking
    Machines Corporation software that you are running. |#

;;; Connection Machine Software, Release 6-0 with *Lisp 6.0 and
;;; CM Graphics 2.0
;;;
;;; Copyright (C) 1990 by Thinking Machines Corporation.
;;; All rights reserved.

#|  Finally, Lucid Common Lisp loads the file lisp-init.lisp from your
    home directory, if you have already created such a file. |#

;;; Loading source file "/users/admin1/massar/lisp-init.lisp"
```

```
#|   Massar's lisp-init.lisp file includes commands to display information
     about current the state of the Lucid and *Lisp compilers. |#

The Lucid Compiler is in DEVELOPMENT mode
The *Lisp Compiler is ON
The *Lisp Compiler Code Walker is ENABLED
The *Lisp Compiler safety level is set to FULL SAFETY
The *Lisp Compiler warning level is HIGH
The *Lisp Compiler *use-always-instructions* mode is OFF


#|   At this point, the *Lisp software is loaded, and you will see the
     Lisp prompt ">", indicating that *Lisp is ready to accept commands.
     The first thing to do is type the command (*lisp), to select the
     *Lisp package: |#

> (*lisp)
Default package is now *LISP.


#|   Now type the command cm:finger to see if any Connection Machine
     hardware is available. In this case, the output of cm:finger is: |#

> (cm:finger)
CM           Seqs  Size     Front end  I/F      User      Idle    Command
-------------------------------------------------------------------------
LANA         ---   ---      cyclops     2       (nobody)
LANA         ---   ---      cyclops     0       (nobody)
         256K memory, 32-bit floating point
         framebuffers on sequencers 0 1 (seqs 0 1 are free)
         CMIOC on sequencer 0 (seq 0 is free)
            2 free seqs on LANA -- 0 1 -- totalling 8K procs


#|   This message shows that the CM named 'LANA' has two free sequencers.
     To attach to LANA and initialize *Lisp, we type: |#

> (*cold-boot)

Not attached.  Attaching...
;;; Loading source file "/cm/configuration/configuration.lisp"
4096
(64 64)
>


#|   At this point, *Lisp is loaded, CM hardware is attached, and both
     *Lisp and the CM have been initialized. *Lisp is now ready for you to
     begin loading and running your own *Lisp programs. |#
```

D.2 *Lisp Simulator Startup Session

```
#|  As with the hardware version of *Lisp, type the UNIX-level command
    that loads the *Lisp software. In this case, it is: |#

% /usr/local/starlisp-simulator


#|  Sun Common Lisp displays its version and copyright information: |#

;;; Sun Common Lisp, Development Environment 3.0.5 (Rev 01), 30-Aug-90
;;; Sun-4 Version for SunOS 4.0
;;;
;;; Copyright (c) 1985, 1986, 1987, 1988 by Sun Microsystems, Inc., All
Rights Reserved
;;; Copyright (c) 1985, 1986, 1987, 1988 by Lucid, Inc., All Rights Re-
served
;;;
;;; This software product contains confidential and trade secret
;;; information belonging to Sun Microsystems.  It may not be copied
;;; for any reason other than for archival and backup purposes.


#|  The the "CM Initializations" file is loaded, displaying the current
    *Lisp patch level: |#

;;; Loading source file "/cm/patch/initializations.lisp"
;;; *Lisp Patch Level 0


#|  Your "lisp-init.lisp" file is loaded, if you have created one: |#
;;; Loading source file "/users/admin1/massar/lisp-init.lisp"
The Lucid Compiler is in DEVELOPMENT mode


#|  Now the simulator is loaded and ready for use. To initialize it, type
    the *Lisp command to select the *Lisp software package: |#
> (*lisp)
Default package is now *SIM.


#|  And type *cold-boot to initialize *Lisp. |#
> (*cold-boot)
Thinking Machines Starlisp Simulator.  Version 18.0
1
(8 4)
>


#|  The *Lisp simulator is now loaded and ready for you to begin loading
    and running your own *Lisp programs. |#
```

Index

Language Index

This index lists all *Lisp and Common Lisp language elements used in this document. Page numbers printed in **bold face** type contain definitions, important examples, or central concepts related to a particular function or macro. Operators that have no bold-faced page numbers are only used in examples and are not defined in this document.

Definitions of all *Lisp functions and macros are available in the **Lisp Dictionary*.

Definitions of all Common Lisp functions, macros, and special forms are available in *Common Lisp: The Language*, by Guy L. Steele (Burlington, Massachusetts: Digital, 1990).

Symbols & Numbers

!!, **22**, 26, 29, 46
+!!, 23, 25, **30**, 32, 33, 39, 63, 65, 69, 79, 80, 81, 85, 86, 89, 91, 92, 97, 120, 121
–!!, 25, 39, 42, 100
***!!**, 7, **30**, 32, 79, 85, 86, 89, 92, 93, 106, 121
/!!, 7, 26, **30**, 60, 63, 70
/=!!, 106
=!!, 99
<!!, 14, 51, 100, 101, 106, 122, 123
<=!!, 14, 122, 123
>!!, 14, 122, 123
1+!!, 7, 14, **30**, 121, 122, 123
1–!!, 14, 122, 123

A

***all**, 40, 106
allocate!!, **66**, **114**
allocate–list–pvar, definition, **109**
allocate–processors–for–vp–set, **108**, 109
***and**, **47**
apply, 124
array–to–pvar, **48**
automata–grid, definition, **5**, **119**

B

break, 61, **62**

C

cadr, 14, 120, 122
car, 14, 120, 122
case, 40
char–int!!, 95
cm:attach, **135**, 136
cm:attached, **136**
cm:detach, **19**, **137**
cm:finger, **4**, **136**, 141
cm:time, **89**, 89—91, **137**
coerce!!, 88
***cold–boot**, **4**, 5, 19, 27, **28**, **38**, 45, **52**, 52, 55, **62**, **64**, 66, 70, 135, 141, 142
compile, **34**, 60, 69, 75, 91, 92, 93, 94, 95
compile–file, **34**, 75
compilep, **80**
compiler–options, **78**
compiling, **80**
complex!!, **26**
cond, 40
cond!!, 14, 122, 123

C, cont.

create–segment–set!!, 114
current–automaton, definition, 123
current–cm–configuration, **28**, 52, 55
current–vp–set, **56**, 56

D

deadp, definition, **17**, **121**
deallocate–processors–for–vp–set, **108**
declare, 69, 79, **86**, 92, 93, 94, 95, 99, 101
default–vp–set, **56**, 56
defaultp, definition, **56**
defautomaton, definition, **122**
defmacro, **33**, 109, 122
***defstruct**, 26, 105, 114
***defun**, **33**
defun, 5, 6, 7, 8, 11, 13, 14, 15, 17, **32**, 44, 60, 69, 85, 86, 87, 89, 90, 91, 92, 93, 94, 95, 99, 100, 101, 102, 105, 107, 110, 120, 121, 122, 123, 124
***defvar**, 5, **23**, 24, 26, 39, 40, 45, 47, 48, 50, 51, 53, 65, 66, **84**, 90, 100, 104, 106, 108, 119
defvar, 8, 12, 48, 66, 85, 119, 122, 123
def–vp–set, **53**, 108
describe–pvar, 69
dolist, 14, 120
dotimes, 8, 11, 65, 86, 90, 110, 121, 124

E

ecase, 13, 40, 121
enumerate!!, **40**, 94, 107
eq, **56**
evenp!!, 7, 25, **39**, 40, 47, 61, 121, 123
exp!!, **30**

F

find!!, 113
float!!, 26, **30**
floor!!, 7, 121, 123
format, 110
front–end!!, **26**

G, H, I

grid, **6**, 45, 120
if, 40, 93, 94
if!!, 7, 14, **39**, 40, 47, 99, 100, 121, 122, 123
immediate–error–if–location, **78**
in–package, 119
init, definition, **14**, **120**
initialize, definition, **124**
int–char!!, 26, 95
interpreter–safety, **62**, 63

L

lambda, 122
lcl:quit, **19**
length!!, 113
***let**, 13, 14, **23**, 65, **86**, 99, 100, 101, 102, 105, 106, 109, 110, 121, 122, 123
let, 48, 80, 101, 105, 109, 110
***light**, **114**
***lisp**, **4**, 19, 141, 142
list, 55, 109, 120, 122
list–start–processor, definition, **108**
load, **34**
loop, 66

M

macroexpand–1, **80**
make–array!!, 26, 113
***max**, **47**, 103
max, 109
***min**, **47**
minimum–size–for–vp–set, **108**, 109
mod!!, 8, 50, 104, 120
moore–count, definition, **13**, **121**

N

neighbor–count, definition, **13**, **121**
neighborhood, definition, **12**, **119**
neumann–count, definition, **13**, **120**
***news**, **44**
news!!, 12, 13, **44**, 120, 121
next–power–of–two–>=, **108**, 109

N, cont.

nil!!, **23**, 25, 47
not!!, 38
number–of–elements, definition, **108**
number–of–processors–limit, **28**, 42, 52, 55

O

oddp!!, 39
one–step, definition, 7, **14**, **121**, **122**
***or**, **47**
or!!, 14, 99, 106, 122, 123

P

pbp, definition, **100**
plusp!!, 109
position!!, 113
ppme, **80**
ppp, 5, **24**, 25, 30, **31**, 33, 39, 40, 42, 46, 47, 48, 49, 50, 51, 100, 101, 102, 104, 106, 110, 120
ppp!!, **101**
pprint, **80**
pref, 6, **24**, 45, 97, 99, 103, 104, 110, 120
pref!!, **42**, 100, 104, 105, 106
pref!!–many–collisions, definition, **105**
pretty–print–pvar, **5**
  See also **ppp**
print, 48
print–array, 48
print–lists, definition, **110**
***proclaim**, **84**, 89, 90, 93
proclaim, **84**, 85
progn, 109, 122
***pset**, **42**, 94, 105, 106, 107, 109
push, 66
pvar–to–array, **48**

Q

quit. *See* **lcl:quit**, **sys:quit**

R

random!!, 6, 8, 25, **30**, 47, 51, 53, 86, 110, 120
random–grid, definition, **8**, **120**
rank!!, **51**, 105, 107
read–cell, definition, **6**, **120**
realpart!!, 85
***room**, 68
room, 68
run, definition, **8**, **121**

S

safety, **77**, 78
scan!!, **49**, 98, 99, 102, 103, 105, 106, 107, 108, 109, 114
segment–set–scan!!, 114
segmented–news!!, definition, **102**, **103**
self–address!!, **25**, 26, 30, 31, 32, 33, 39, 40, 42, 47, 50, 51, 61, 70, 99, 100, 102, 103, 104, 105, 106, 110
serial–one–step, definition, **11**
***set**, 6, 8, **24**, 33, 38, 39, 40, 65, 80, 85, 90, 93, 94, 99, 101, 102, 103, 106, 109, 120
set–cell, definition, **6**, **120**
set–cells, definition, **14**, **120**
set–grid, definition, **6**, **8**, **120**
set–vp–set, **55**, 56
***setf**, 6, **24**
  applied to **pref**, 6, **24**, 45, 99, 103, 104, 120
setf, 6
setq, 30, 77, 78, 79, 80, 91, 122, 123
setup, definition, **123**
sideways–aref!!, 113
***sideways–array**, 113
signum!!, **13**, 17, 121
sin!!, **30**
slc::*show–expanded–code*, **79**, 80
sort!!, **51**
spread!!, **51**
substitute!!, 113

S, cont.

***sum**, 17, **47**, 61, 62, 63, 105, 109, 121
sys:quit, **19**

T

t!!, **23**, 25, 47, 102, 103
test–lists, definition, **110**
test–very–long–add, definition, **101**
the, 80, 81, **87**, 94
total–number–of–states, definition, **8**, **119**
trace–stack, **67**, 67—68

U

unless, 40

V

values, 100, 101
very–long–add!!, definition, **99**
view, definition, **5**, **120**
view–step, definition, **15**, **121**

W, X, Y, Z

***warm–boot**, **61**, 62, 65, 66, 69, 70
warning–level, **77**, **91**
***when**, **38**, 39, 40, 47, 51, 61, 101, 103, 106, 109
when, 40
with–css–saved, 40
***with–vp–set**, **55**, 109, 110
zerop, 17, 121
zerop!!, 14, 50, 99, 104, 106, 122, 123

Concept Index

This index lists all *Lisp and Common Lisp concepts used in this document. Page numbers printed in **bold face** type contain definitions or important examples of a particular concept.

Items printed in *italic* type are titles of documents in the Connection Machine Documentation set, or titles of related books. A list of all referenced documents can be found in the *About This Manual* section, under the heading "Related Documents."

---

*Lisp

*Lisp, **1**, 128, **131**, 131–133
- !! and *, naming convention, **30**
- batch mode, **133**
- compiled code, displaying, **79**, 79–81
  - Symbolics commands for, **81**
- compiler, **75**, 75–76, **132**
  - safety level, **77**
  - warning level, **77**
  - warning messages.
    - *See* compiler warning messages
- compiler options, **77**, 77–78
- compiling, **34**, **75**, 75–76, 89–91
- data parallelism. *See* data parallelism
- data types. *See* pvar data types
- declaring.
  - *See* type declaration (separate topic)
- error checking
  - compiler, **77**
    - *See also* *safety*, *warning–level*
  - interpreter, **62**
    - *See also* *interpreter–safety*
- exiting, **19**
- front–end Lisp environment, **132**
- hardware version, **4**
- initializing, **4**, 19, 27, **52**
- interpreter, 62, **132**
- language, **1**, 21, 128, **131**

*Lisp, cont.
- loading software, **4**
- on the CM, **131**
- on-line code examples, **18**, 97, **115**
- selecting package, **4**, 19
- simulator version, **4**, 45, 52, **132**
- timesharing, **133**
- timing, **89**, 89–91, **137**
- type declarations.
  - *See* pvar type declarations
- using, **5**

*Lisp compiler. *See* *Lisp, compiler

Lisp Dictionary*, **xi, 30, 33, 40, 65, 68, 75, 78, 81, 101, 108, 113, 115

*Lisp in batch. *See* *Lisp, batch

*Lisp interpreter. *See* *Lisp, interpreter

*Lisp simulator. *See* *Lisp, simulator version

*Lisp under timesharing.
- *See* *Lisp, timesharing

A

aborting from the debugger.
- *See* debugger, aborting from

active processors, **38**, **130**
- *See also* currently selected set

addresses, of processors.
- *See* processor addresses

allocating a pvar. *See* pvar, allocating

A, cont.

array
  See also "sideways" array pvars
  pvar data type, **26**, 83, 113
    *Lisp operations for, **113**
  pvar type specifier, **83**
array/pvar conversion functions, 46, **48**
arrays, and pvars, similarities, **27**
attaching to a CM. *See* CM, attaching
automata. *See* cellular automata

B

batch mode *Lisp. *See* *Lisp, batch mode
bit-array pvars, **113**
  See also array, pvar data type
boolean
  pvar data type, **25**, 82
  pvar type specifier, **82**
bulk data movement.
    See array/pvar conversion functions

C

calling parallel functions.
    See parallel functions, calling
calling parallel macros.
    See parallel macros, calling
cell. *See* cellular automata
cellular automata, **2**, 2–3
  9 Life. *See* 9 Life automaton
  cells, **2**, 5
  defining, **7**, 7–9, 10–16
  grid, **2**, 3, 5, 12
  Life. *See* Game of Life
  neighborhoods, **10**
    Moore, **10**, 13, 16
    Von Neumann, **10**, 13, 16
  neighbors, 2, **10**
cellular automata, cont.
  rules, **2**, 7, 10
  states, **2**, 3
  typical ending conditions of, **9**
changing a value of a pvar. *See* pvar
character
  pvar data type, **26**, 83
  pvar type specifier, **83**
clear the stack,
    *Lisp operators that automatically, **65**
CM, ix, **1**, 4, 5, 11, **127**, 127–133
  See also Connection Machine system
  attaching, **4**, 27, **135**
  cold booting, **27**, **52**
    and active processors, **38**
  current state, **28**
    See also configuration variables
  data parallelism of. *See* data parallelism
  detaching, **19**, **137**
  FEBI. *See* FEBI
  finding a CM to use. *See* CM, fingering
  fingering, **136**
  front end. *See* front end
  front panel LEDs. *See* LEDs, front panel
  H-bus cable. *See* H-bus cable
  initializing. *See* CM, cold booting
  memory
    and pvars, **22**, 27
    stack and heap, **65**
  Nexus. *See* Nexus
  parallel instruction set. *See* Paris
  resetting the state of.
      See CM, warm booting
  section. *See* section
  sequencer. *See* sequencer
  warm booting, **61**
CM Parallel Instruction Set. *See* Paris
CM Parallel Instruction Set (Paris), **xi**, 114

C, cont.

CM processors, 5, 11, 12, **129**
  address of. *See* send address
  arrangement, **27**
    See also processor grid
  communication.
    See processor communication
  selection and deselection.
    See processor selection
  value of a pvar for, 21, **22**, 25
CM System User's Guide, ix, **xi**, 4, 19, 135
CM Technical Summary, ix, **xi**
code examples, on-line. *See* *Lisp, online code examples
cold booting. *See* ***cold–boot**; CM, cold booting
command
  editor, for compiling code. *See* editor commands for compiling code
  for loading *Lisp. *See* *Lisp, loading software
Common Lisp
  language, **ix**, 1, 21, 131
  tutorials, **ix**, xi
Common Lisp: The Language, ix, **xii**
communication
  See also processor communication
  between front end and CM, **46**, **128**
  "global" operators, 46, **47**
compiler
  *Lisp. *See* *Lisp, compiler
  safety level. *See* *Lisp, compiler safety level
  warning level. *See* *Lisp, compiler warning level
compiler error checking. *See* *Lisp, error checking
compiler options. *See* *Lisp, compiler options
compiler warning messages, **91**, 91–95
  can't determine type returned by, **92**
compiler warning messages, cont.
  can't find type declaration, **91**, **92**
  does not compile special form, **93**
  does not understand how to compile, **94**
  function expected a...argument but got, **95**
compiling *Lisp code. *See* *Lisp, compiling
complex
  pvar data type, **26**, 83
  pvar type specifier, **83**
configuration
  See also processor grid
  *Lisp operators. *See* ***cold–boot**; VP sets
  custom. *See* VP sets
  of processors, **27**, 28
configuration variables, **28**
Connection Machine Parallel Instruction Set (Paris), **xi**, 114
Connection Machine Parallel Instruction Set (Paris). *See* Paris
Connection Machine philosophy.
  See The Connection Machine
Connection Machine system, **ix**, xii, 1
  See also CM
Connection Machine, The. **xii**
"Connection Machine".
  See Connection Machine system
conversion
  between arrays and pvars.
    See array/pvar conversion functions
  of scalars to pvars. *See* scalar promotion
Conway, John H., **2**
copying a pvar. *See* pvar, copying
copying data across the processor grid.
  See scanning, spreading
creating a pvar. *See* pvar, defining
creating VP sets. *See* VP sets
cumulative parallel operations. *See* scanning
current VP set. *See* VP sets
currently selected set
  See also active processors
  of processors, **38**, **130**

D

data parallel processing, **127**
  CM model, **128**
  examples, **128**
data parallel programming, **128**
data parallelism, **6**, **25**, 37, 128
data types
  of pvars. *See* pvar data types
  parallel. *See* pvar data types
  scalar. *See* scalar data types
deallocating a pvar. *See* pvar, deallocating
debugger, 59, **70**
  aborting from, **61**
  commands, **71**
  display, **70**, **71**
  messages from. *See* error messages
declarations. *See* type declarations
default VP set. *See* VP sets
defined–float
  pvar data type, **26**, 82
  pvar type specifier, **82**
defining a pvar. *See* pvar, defining
defining parallel functions, **32**, 32–33
  See also parallel functions
defining parallel macros, **33**
  See also parallel macros
defining VP sets. *See* VP sets
deselection of processors.
    See processor selection
detaching from a CM. *See* CM, detaching
displaying compiled code.
    See *Lisp, compiled code, displaying

E

editor commands, for compiling code, **34**, 76
enumerating selected processors, **40**
error checking. *See* *Lisp, error checking
error messages, **60**, **64**
  break message, **61**
  "bus error," **64**, 69
  executing code without CM attached, **64**
  executing code without cold booting, **64**
  FEBI error message. *See* FEBI
error messages, cont.
  from stack-tracing facility, **67**
  "is not a pvar," **69**
  obscure error messages, **70**
  parallel division by 0, 60, 63, **71**
  running out of CM heap memory, **66**
  running out of CM stack memory, **65**
  running out of temporary pvars, **66**
  running out of VP sets and geometries, **66**
  sequencer error message. *See* sequencer
  "timed out on safe FIFO read", **64**
  typing mistake, **60**
error recovery, **61**, 61–62
  importance of warm booting, **61**
examples, on-line.
    See *Lisp, on-line code examples
exiting *Lisp. *See* *Lisp, exiting
exiting from the debugger.
    See debugger, aborting from

F

FEBI (front end bus interface), **129**
  error message about, **4**
finding a CM to use. *See* CM, fingering
flexible VP set. *See* VP set, flexible
floating-point pvars. *See* defined-float
front end, 4, 26, 27, 37, 46, 48, **128**
front end bus interface card. *See* FEBI
front panel LEDs. *See* LEDs, front panel
front-end
  pvar data type, **26**, 83
  pvar type specifier, **83**
front–end/CM communication.
    See communication
functions
  parallel. *See* parallel functions
  user-defined. *See* parallel functions
funnels for data.
    See "global" communication
    operators

G

Game of Life, **2**, 10
GC, warning.
    See warning messages, GC warning
general
  pvar data type, **26**, 83
  pvar type specifier, **83**
"get," parallel. *See* parallel "get"
getting a value from a pvar. *See* pvar
"global" communication operators, 46, **47**
global parallel variable. *See* permanent pvar
global pvars
  See also permanent pvar; pvars, global
  defined by **allocate!!**, **114**
global type declarations.
    See type declarations, global
global variables, in *Lisp.
    See permanent pvar
grid
  cellular automata. *See* cellular automata
  processor. *See* processor grid
  scanning. *See* scanning
grid communication. *See* processor
    communication

H

H-bus cable, **129**
heap, CM. *See* CM memory, stack and heap

I

:include–self, argument to **scan!!**, **49**, **100**
initial grid configuration
  See also default VP set
  set by ***cold–boot**, **52**
initializing the CM. *See* CM, cold booting
integer pvars. *See* signed–byte, unsigned–byte
interpreter error checking.
    See *Lisp, error checking
interpreter, *Lisp. *See* *Lisp interpreter

K

keystroke, editor, for compiling code.
    See editor commands for
    compiling code

L

LEDs, front panel, **114**
  See also ***light**
Life. *See* Game of Life
lion, statue of, **38**
Lisp. *See* Common Lisp
Lisp/Paris. *See* Paris
list, data structure, 1
  lack of pvar data type for, **108**
  using *Lisp tools to simulate lists, **108**
"list" pvar, **108**
lists. *See* list
Loading *Lisp.
    See *Lisp, loading software
local parallel variables. *See* pvar, local
local pvar. *See* pvar, local
local variables, declaring the type of, **86**

M

macros
  parallel. *See* parallel macros
  user-defined. *See* parallel macros
massively parallel processing, **127**
memory
  CM. *See* CM memory
  running out of. *See* error messages
messages. *See* processor communication
Michelangelo, **38**, 39
modifying a value of a pvar. *See* pvar
moving data to/from front end.
    See array/pvar conversion functions

N

neighbor. *See* cellular automata
neighborhood. *See* cellular automata
news communication.
    See grid communication
"NEWS" communication, historical note, **44**
Nexus, **129**
9 Life automaton, **10**, 14, 17, 123
numeric pvars
  complex. *See* complex pvar data type
  float. *See* floating-point pvars
  integer. *See* integer pvars

O

on-line code examples.
    See *Lisp, online code examples
options, compiler.
    See *Lisp, compiler options

P

package, *Lisp.
    See *Lisp, selecting package
parallel data transformations, **49**
  See also
    parallel prefixing;
    scanning;
    sorting;
    spreading
parallel equivalent,
    of a Common Lisp function, **30**
parallel functions
  calling, **33**
  predefined, **30**
  user-defined, **32**
    arguments, declaring the type of, **86**
    declaring the type of, **85**
parallel "get", **42**
  See also router communication
parallel instruction set. *See* Paris
parallel macros
  calling, **34**
  user-defined, **33**
parallel prefixing. *See* scanning
parallel processing, **127**
  See also serial processing
parallel "send", **42**
  See also router communication
parallel sort. *See* sorting
parallel variable, 1, 5, **21**
  See also pvar
parallelism
  data. *See* data parallelism
  of CM, 5, **127**
Paris, 76, 79, 128, 132, **135**
permanent pvar. *See* pvar, permanent
physical processors, 53, **131**
predefined pvars. *See* pvar, predefined
pretty-print-macroexpand. *See* **ppme**
printing a pvar. *See* pvar, printing
processor communication, **41**, **130**
  grid communication, **43**, 102, 130
    *Lisp operators for, **44**
  router communication, **41**, 102, 104, 108, 130
    *Lisp operators for, **42**
processor grid, **27**
  See also configuration; configuration variables; VP sets
  and grid communication, **43**
  and pvar shapes, 27, **29**, 52
  and VP sets, 27, 29, **53**
  setting the size and shape of, **27**, **52**
processor selection, **38**, **130**
  *Lisp operators for, 38, **40**
  and deselection, **38**
  and enumeration. *See* enumeration
processors, **127**
  CM. *See* CM processors
  memory of, **129**
  physical. *See* physical processors
  virtual. *See* virtual processors; VP sets
promotion, of scalars to pvars. *See* scalar promotion

P, cont.

pvar, 1, 5, **21**
- allocating, deallocating, **22**
- array. *See* array, pvar data type
- bit-array. *See* bit-array pvars
- boolean. *See* boolean, pvar data type
- changing a value of, 6, **24**
- character. *See* character, pvar data type
- complex. *See* complex, pvar data type
- copying, **24**
- data types, 21, **25**, 25–26, **81**, 81
- declaring the type of, **84**
- defined–float. *See* defined–float, pvar data type
- defining, 5, **22**, 23
- front–end. *See* front–end, pvar data type
- general. *See* general, pvar data type
- global, **114**
- global variables. *See* pvar, permanent
- "list" pvar, **108**
- local, **23**, 86
  - declaring the type of, **86**
- permanent, 5, **23**, 84, 108
  - declaring the type of, **84**
- predefined, **23**
- printing, 5, **24**, **31**, 31–32
- reading a value from, 6, **23**
- sequence. *See* sequence pvars
- setting the value of, 6, **24**
- signed–byte. *See* signed, pvar data type
- similarity to arrays, **27**
- size and shape of, 27–29, **29**
- sorting the values of, **51**
- structure. *See* structure, pvar data type
- temporary, **22**
- type declarations, 34, **81**, 81–87
  - *See also* type declaration (separate topic)
- type specifiers, **82**, 82–83
- undeclared
  - default to general type, **83**
  - will not compile, **88**
- undefining all pvars, **62**

pvar, cont.
- unsigned–byte. *See* unsigned–byte, pvar data type
- using VP sets to define, **53**
- values of, **21**
- vector. *See* vector pvars

pvar/array conversion functions. *See* array/pvar conversion functions

Q, R

quitting *Lisp. *See* *Lisp, exiting

rational numbers, lack of pvar data type for, **7**

reading a value from a pvar. *See* pvar

recovering from errors. *See* error recovery

router, **41**, **130**
- *See also* processor communication

router communication. *See* processor communication

rule. *See* cellular automata

running *Lisp code
- compiled. *See* *Lisp, compiler
- in batch. *See* *Lisp, batch
- interpreted. *See* *Lisp, interpreter
- simulated. *See* *Lisp, simulator version
- under timesharing. *See* *Lisp, timesharing

S

safety level, compiler. *See* *Lisp, compiler safety level

scalar
- data types, **21**, 25
- promotion to pvars, 23, 25
- values, **21**, 22
- variables, declaring the type of, **85**, **86**

scanning, **49**, 97–112
- examples of
  - adding very large integers, **97**, 97–101
  - defining new parallel data structures, **108**, 108–111
  - defining segmented functions, **102**, 102–104

S, cont.

scanning, cont.
  examples of, cont.
    using the router efficiently, **104**, 104–107
  segmented, **50**
  with segment sets. *See* segment set objects
section, CM, **130**
segment set objects, **114**
segmented pvars. *See* scanning, segmented
selected set, of processors. *See* currently selected set
selected VP set. *See* current VP set
selecting VP sets. *See* see VP sets
selection of processors. *See* processor selection
send address, **25**
"send," parallel. *See* parallel "send"
sequence pvars, **113**
  See also array, pvar data type
sequencer, **130**
  error message about, **4**
serial processing, **127**
  See also parallel processing
serial programming, compared with parallel, 11
setting a pvar. *See* pvar, copying
setting a value of a pvar. *See* pvar
"sideways" array pvars, **113**
signed–byte
  pvar data type, **25**, 82
  pvar type specifier, **82**
simulator. *See* *Lisp, simulator version
sorting, **51**
spreading, **50**
stack
  clearing. *See* clear the stack
  CM. *See* CM memory, stack and heap
Starlisp. *See* *Lisp
Starting *Lisp. *See* *Lisp, loading software
state. *See* cellular automata
statue of lion. *See* lion, statue of
structure
  pvar data type, **26**, 83, 105, **114**
    example of, **105**
  pvar type specifier, **83**

T

temporary pvar. *See* pvar, temporary
timesharing *Lisp. *See* *Lisp, timesharing
timing *Lisp code. *See* *Lisp, timing
tracing stack memory use, **67–68**
transformations, on pvars. *See* parallel data transformations
type declaration
  and general pvars, **26**, 88
  and the compiler, 34, **76**, **87**, 87–88
  complete guidelines, **75**
  effect of. *See* type declaration and the compiler
  examples of, **84**, 89
  for pvars. *See* pvar type declarations
  global, via **proclaim**, **84**
  in *Lisp, 35, 76, **82**
  in-line, via **the**, **87**
  local, via **declare**, **86**
  using, **84**
type specifiers, for pvars. *See* pvar type specifiers

U

undeclared pvars. *See* pvar, undeclared
undefining all pvars, **62**
unsigned–byte
  pvar data type, **25**, 82
  pvar type specifier, **82**
using *Lisp. *See* *Lisp, using

V

value(s), of a pvar.
See pvar, values of
variable
configuration. *See* configuration variables
parallel. *See* pvar
VP set. *See* VP set variables
vector pvars, **113**
See also array, pvar data type
virtual processor sets. *See* VP sets
virtual processors, **53**, **131**
effect on CM operation, **53**
VP set variables, **56**
VP sets, 29, **53**, **131**
and processor grids, 27
default and current, **54**
defining, **53**
example of, **108**
flexible, **108**
selecting, **55**
used to define pvar shapes, **53**
using, **54**
why VP "sets," **54**
VP's. *See* virtual processors

W

warm booting.
See *warm–boot;
CM, warm booting
warning level, compiler.
See *Lisp, compiler warning level
warning messages
from the *Lisp compiler, 60
See also compiler warning messages
from your Lisp system, **59**
GC warning, **59**
writing a value of a pvar. *See* pvar

Thinking Machines Corporation
245 First Street
Cambridge, MA 02142-1264
(617) 234-1000