Thinking Machines Technical Report 86.15

The Essential *Lisp Manual

Release 1, Revision 3

With Programming Examples
for the Connection Machine® System

April 1986

Acknowledgments

*LISP is a language designed to run on the Connection Machine system, Thinking Machines Corporation's data parallel computer. The design and implementation of the language, as well as the creation and revision of this manual, are the result of the efforts of many people at Thinking Machines. *LISP is the result of four years of language development. The original language, URDU, was designed by Cliff Lasser in 1982 when the Connection Machine system was still in its design phase at M.I.T. While the hardware was being built at Thinking Machines, URDU evolved into SIMPL. Based on many users' experiences with SIMPL on the Connection Machine hardware, *LISP emerged.

- Cliff Lasser and Steve Omohundro were the primary designers and authors of the *LISP language and this manual.
- Other people who contributed to the design were Guy Blelloch, Brewster Kahle, JP Massar, John Rose, and Jim Salem.
- Cliff Lasser and George Robertson implemented the hardware version.
- JP Massar implemented the simulator.
- Janet MacLaren, JP Massar, and Charles Perkins edited and produced this manual.
- Thanks also to Danny Hillis and Guy Steele for their continual support and assistance.

Contents

List of Figures

Chapter 1

Introduction

Welcome to *LISP!! This language (pronounced *star lisp*) is an extension of COMMON LISP, which runs on Connection Machine data parallel computers. This ESSENTIAL *LISP manual describes the fundamental instruction set used in *LISP. More advanced features are described in the COMPLETE *LISP manual.

This manual is organized into the following chapters:

1. **Introduction**
   This chapter. It provides an overview of the ESSENTIAL *LISP manual's organization, and explains the significance of the different typefaces used throughout the manual.

2. **Connection Machine Computers and *LISP**
   This chapter provides an overview of the Connection Machine computer and of the language ESSENTIAL *LISP, including:

   - a description of the name conventions and conceptual structure of the language
   - example expressions that suggest the feel of the language
   - a description of the conventions used in presenting language functions
   - an introduction to the Lisp constants that specify the configuration parameters of the Connection Machine system for use by programs

3. **The Pvar Data Structure**
   This chapter precisely defines the data structure known as a *pvar* and the operations that create pvar instances and act on them.

4. **Processor Selection**
   This chapter describes the mechanisms for selecting the set of processors that will perform an operation.

5. **!! Functions**
   This chapter describes the functions for comparing and combining pvars numerically and logically, and describes the mechanisms for defining new functions of this type.

6. **Parallel Global Memory References**
   This chapter discusses the mechanisms for moving data between Connection Machine processors.

7. **Global Operations**
   This chapter describes operations for combining data globally from all Connection Machine processors.

8. **Actually Using the Hardware**
   This chapter describes the process of actually using the hardware, including procedures for logging in to and resetting the Connection Machine system.

9. **Potentially Troublesome Situations**
   This chapter describes potential ambiguities and trouble spots in the ESSENTIAL *LISP language definition.

10. **Appendix A - Some Example Programs**
    This chapter provides some sample programs written in *LISP.

11. **Appendix B - List of ESSENTIAL *LISP Commands**
    This chapter summarizes all the commands described in ESSENTIAL *LISP and can be used as a quick reference guide. Commands are grouped by functionality, rather than by name; so if users know what they want to do, but not which *LISP command to use in order to do it, then this is the place to go for help.

1.1 The Use of Different Typefaces in This Manual

Throughout the ESSENTIAL *LISP manual, actual code (including actual names of functions) will generally appear in the following typeface:

```
(cons a b)
```

Names that stand for pieces of code (metavariables) will generally be written in *italics.* Note that in function descriptions, the names of the parameters will appear in italics for expository purposes.

Chapter 2

Connection Machine Computers and *LISP

2.1 The Connection Machine Computer Organization

A Connection Machine data parallel computer consists of a large number of simple processors. Each has some associated local memory and is integrated into a highly connected communications network. A typical configuration might have 65,536 processors with 4,096 bits of memory each. Typical applications utilize data structures that have components spanning many Connection Machine processors. The goal of *LISP is to provide a language for creating and manipulating these structures in parallel.

There are several intuitive images of the Connection Machine computer organization that are useful to keep in mind when thinking about programs. Perhaps the simplest picture represents the memory of a single processor as a column with a slot for each bit location. The whole machine is represented as a rectangular grid in which successive columns represent successive processors. Thus for the machine configuration on the following page, imagine a rectangle with 65,536 columns and 4,096 rows (see Figure 2.1).

The position of a bit in the memory of its processor is known as its location. A region of contiguous bits in each processor's memory is known as a field. A field's location is the location of its first bit. Figure 2.1 provides a graphical image of these concepts; all memory bits contained in a one bit field at location 1 are labelled "f1", and all bits contained in a length 5 field at location 4 are marked "f2". In *LISP, a data structure containing the location, length, and type of a Connection Machine field is called a *pvar* (for "parallel variable"). A pvar corresponding to the field "f2" marked in Figure 2.1 would be a Lisp object containing the numbers 4 and 5, as well as other information.

Because the processors are tied together by a communications network, it may be helpful to extend the above picture mentally. One useful communication configuration considers the processors to be organized in a two-dimensional grid. In this case, one might picture the machine as a rectangular parallelepiped; the top face represents the grid of processors, and the column beneath each square of the grid represents the bits in the corresponding processor's memory. This image is convenient because it utilizes well developed three-dimensional intuition in thinking about data movement.

A more realistic representation organizes the processors on the corners of a high-dimensional hypercube. Unfortunately, it is very difficult to completely picture this organization.

With *LISP, one is able to specify the number of dimensions, and the sizes of those dimensions, that the Connection Machine system is to simulate. Due to its very high-bandwidth, flexible hy-

	Processor Address									
Location	0	1	2	3	4	5	6	. . .	65,533	65,534	65,535
0											
1	f1	f1	f1	f1	f1	f1	f1	. . .	f1	f1	f1
2											
3											
4	f2	f2	f2	f2	f2	f2	f2	. . .	f2	f2	f2
5	f2	f2	f2	f2	f2	f2	f2	. . .	f2	f2	f2
6	f2	f2	f2	f2	f2	f2	f2	. . .	f2	f2	f2
7	f2	f2	f2	f2	f2	f2	f2	. . .	f2	f2	f2
8	f2	f2	f2	f2	f2	f2	f2	. . .	f2	f2	f2
9											
.											
.											
.											
4093											
4094											
4095											

Figure 2.1: A conceptual model of a 4096 bit by 65,536 processor Connection Machine computer. (f1 and f2 represent fields)

percube router network, the Connection Machine system is able to efficiently simulate almost any interconnection pattern. A thorough description of the system can be found in *The Connection Machine* by W. Daniel Hillis [Hillis 85].

Programs for the Connection Machine system are written in *LISP on a host computer. This host machine is typically a Symbolics 3600 series Lisp Machine or a Digital Equipment Corporation VAX. There are three communication channels between the host computer and the Connection Machine computer:

1. The host generates an instruction stream that is broadcast to all Connection Machine processors (whether a processor executes an instruction depends on its internal state). Most *LISP operations affect the Connection Machine processors in this way.

2. The host may read or write any location in any processor's memory directly. In *LISP, the instructions `pref` and `pref-grid` use this channel (see Section 3.6).

3. There is the so-called "global" channel from the Connection Machine system back to the host which is used, for example, to see if any processor has a 1 stored in a given memory location. The *LISP instructions `*logior`, `*logand`, `*or`, `*and`, `*min`, and `*max` use this channel (see Chapter 7).

2.2 *LISP Terminology

The following are descriptions of terms that are used with a specific meaning in *LISP:

- **Processors**
  The conceptual entities that operate on data in parallel are called *processors.* Often these correspond to actual hardware processors, but sometimes a single hardware processor simulates several conceptual processors. In this case, the simulated processors are referred to as *virtual processors.* This simulation is transparent to the programmer.

- **Cube Address**
  Each processor has a unique *cube address.* The cube addresses of the processors in the Connection Machine system range between zero and the number of processors less 1. On a full sized machine, this would be between 0 and 65,535. The cube address of a processor is independent of how the Connection Machine system is configured at initialization time.

- **Grid Address**
  A specific processor can also be identified by one or more numbers, referred to as the processor's *grid address.* The number of coordinates in a *grid address* is determined by the number of dimensions the Connection Machine system is simulating. For example, one might refer to a processor with a grid address of (3,4,1) in a three-dimensional machine. A two-dimensional machine representing a two-dimensional grid would require two grid address coordinates. Note that the numbering scheme for grid addresses in a one-dimensional machine is *not* necessarily the same as that for cube addresses. ESSENTIAL *LISP requires the user to choose a machine configuration at initialization time and to keep that configuration throughout the program. The COMPLETE *LISP manual describes how to dynamically change the size and shape of the Connection Machine system.

- **Field**
  The most primitive form of data in *LISP is a *field.* A field is a string of contiguous bits in the same memory locations of *each* processor. A field in a processor may contain any valid Lisp object. The different processors of a field may contain different types of values as well. A field exists only inside the Connection Machine processors.

- **Pvar**
  A Lisp object that contains all the information necessary to manage a field is called a *pvar.* This is short for *parallel variable.*

- **Contents of a Pvar**
  The phrase *contents of a pvar* is often used to refer to the values stored in the field in the Connection Machine memory that is described by the pvar.

- **Currently Selected Set**
  Some *LISP operations are only carried out in a subset of the Connection Machine processors. This subset is called the *currently selected set* and is specified by using *LISP instructions, such as `*all`, `*when`, `*cond`, and `*if`.

- !!
  The names of instructions that return pvars as their values end with !! (pronounced *"bang-bang"*). This suffix is meant to look like two parallel lines and indicates that a parallel variable is returned. It is an excellent idea for user-defined functions to obey this convention (though nothing enforces it), because it helps ensure that pvars are produced only in contexts where they can be used. There are a few ESSENTIAL *LISP forms whose names do not end in !!, such as ***when**, ***all** and ***let**, that, nonetheless, may optionally return a pvar.

- *
  All ESSENTIAL *LISP functions that do parallel computation and do not end in !! begin with * (pronounced *"star"*). Note that in COMMON LISP * is used as a prefix operator to denote multiplication, but that it is used in *LISP to denote parallel operations (hence, the name *LISP!).

The functions in ESSENTIAL *LISP that end in !! are:

type-of!!	floatp!!	self-address!!
allocate!!	integerp!!	grid-from-cube-address!!
	numberp!!	cube-from-grid-address!!
logeqv!!	eq!! eql!!	
lognot!! logxor!!	and!! or!!	/=!! =!! <!!
logior!! logand!!	not!! xor!!	>!! <=!! >=!!
+!! -!! *!!		
/!! 1+!! 1-!!	cond!! if!!	
!! random!! sqrt!!	deposit-byte!!	
truncate!! float!!	load-byte!!	pref!! pref-grid!!
min!! max!! mod!!	enumerate!!	pref-grid-relative!!

The remaining functions in ESSENTIAL *LISP do not usually return pvars. Those that act in parallel are annotated with a * prefix. Operations of this type include:

*defvar	*let	*all *when
*deallocate	*let*	*cond *if
	*set	
*defun	*cold-boot	*logior
*funcall	*warm-boot	*logand
*apply		
*min	*and *sum	*pset
*max	*or *product	*pset-grid

Most of these functions are parallel analogs of corresponding Lisp functions or deal with inter-processor communication. Precise definitions of their behavior will be provided in other sections.

Other ESSENTIAL *LISP functions include:

`pvarp`	`dimension-size`	`pref`
		`pref-grid`

For a complete list of ESSENTIAL *LISP commands, see Appendix B.

2.3 Overview of ESSENTIAL *LISP

This section contains some example expressions to give the reader an idea of what *LISP expressions actually look like. Precise definitions of all the functions used occur in other sections of the manual. (Note that Appendix B of this manual will refer the user to page numbers on which these definitions begin.)

2.3.1 Pvars

The following are five sample pvars:

```
(*defvar a)
(*defvar b (!! 5) "This is a documentation string.")
(*defvar c (!! -2.67))
(*defvar d t!!)
(*defvar e (1+!! (self-address!!)))
```

The last four have been initialized with specific values: **b** is a Lisp symbol that points to a pvar containing the integer 5 in each processor; c contains the floating point number -2.67 in each processor; d contains the Lisp symbol T in each processor; and e contains an integer that is the cube address of the next higher processor. See Section 3.1 for a more detailed description of ***defvar**.

The function `pref` can be used to read out some of the above values. The arguments of `pref` are a pvar and a cube address. This is analogous to the COMMON LISP **aref**; the pvar is equivalent to an array and the cube address to the array index.

For example,

```
(pref c 0)
```

returns the Lisp value `-2.67`, since that is what is contained in pvar c in processor 0.

Similarly,

```
(pref d 365)
```

returns the Lisp value T because that is what is contained in pvar f in processor 365. See Section 3.6 for a more detailed description of **pref**.

*LISP uses the COMMON LISP macro **setf** to turn accessor expressions into modifier expressions. For example, there is no function that does the opposite of **pref**. To write into a single processor of a pvar, one would write something like:

```
(setf (pref b 0) 15)
```

The form (pref b 0) would now return 15 because 15 was just stored in pvar b in processor 0.

The following demonstrate arithmetic operations on the example pvars:

```
(*set a (+!! b c))
```

will make the contents of pvar a be the sum of the contents of pvar b and pvar c. Notice that because c contains floating point values, the integers contained in b are properly coerced to floating point and the result in a will be floating point as well. See Section 3.5 for a more detailed discussion of ***set**.

Expressions can be nested:

```
(*set a (-!! b (*!! a (!! 2))))
```

This sets a to the difference of b and 2 times a. This simple expression causes thousands of operations to go on simultaneously! See Section 5.3 for a more detailed discussion of numerical !! functions.

2.3.2 Selection

It may be desirable to do an operation in only the processors with odd cube addresses. In this case, select the subset of all processors consisting of those whose cube addresses (contained in the pvar returned by the function **self-address!!**) end in 1 using ***when**:

```
(*when (=!! (!! 1) (mod!! (self-address!!) (!! 2)))
    (*set a (+!! a b)))
```

or equivalently:

```
(*defvar odd-address-p (=!! (!! 1)  (mod!! (self-address!!) (!! 2))))

(*when odd-address-p
   (*set a (+!! a b)))
```

In another case, it may be desirable to do an operation in the processors that have both an even cube address, and in which the pvar a contains zero. Two natural ways to do this are to (1) use the logic functions to select the correct set:

```
(*when (and!! (not!! odd-address-p)
              (=!! a (!! 0)))
  (*set a (+!! a b))))
```

or equivalently, (2) to nest *when expressions:

```
(*when (not!! odd-address-p)
    (*when (=!! a (!! 0)))
      (*set a (+!! a b))))
```

It might also be advantageous to do an operation with a temporary variable allocated. For example, if a programmer wants to do an operation in all processors whose addresses are divisible by four, he or she might say:

```
(*let ((g (mod!! (self-address!!) (!! 4))))
  (*when (=!! g (!! 0))
     (*set a (+!! a b))))
```

This first creates a temporary variable g and loads it up with the two lowest order bits of the (self-address!!). In all processors in which this is 0, the *set operation is performed.

To perform different operations in processors whose addresses have a remainder of 0, 1, 2 or 3 after dividing by 4, the following might be used:

```
(*let ((g (mod!! (self-address!!) (!! 4))))
   (*cond
      ((=!! g (!! 0)) (*set a (+!! a b)))
      ((=!! g (!! 1)) (*set a (-!! a b)))
      ((=!! g (!! 2)) (*set a (*!! a b)))
      ((=!! g (!! 3)) (*set a (//!! a b)))))
```

This may also be done using:

```
(*let ((g (mod!! (self-address!!) (!! 4))))
  (*set a
    (cond!!
      ((=!! g (!! 0)) (+!! a b)))
      ((=!! g (!! 1)) (-!! a b)))
      ((=!! g (!! 2)) (*!! a b)))
      ((=!! g (!! 3)) (//!! a b)))))
```

There are also *LISP expressions analogous to the Lisp if:

```
(*if (<!! z x)
     (*set y (!! 5))
     (*set y (!! 6)))

(*set y
  (if!! (<!! z x)
        (!! 5)
        (!! 6)))
```

Note that if!! returns a pvar that must be stored into some destination, whereas *if is only executed for side effect.

See Chapter 4 for more detailed descriptions of *when, *cond and *if; Section 6.2 for the definition of self-address!!; Section 3.4 for a discussion of *let; Sections 5.2.1 and 5.2.2 for definitions of the logical functions; and Section 5.4 for a description of if!!.

2.3.3 *Defun

To define functions that can take pvars as arguments or return them as values, use *defun instead of defun. To define a function that takes two pvar arguments and returns their sum, difference, product, or quotient (depending on whether the processor's address has remainder 0, 1, 2 or 3 when divided by 4 in all processors in the currently selected set), use something like the following:

```
(*defun four-function!! (pvar-a pvar-b)
   (*let ((address-bits (mod!! (self-address!!) (!! 4)))
          (answer))
      (*cond
         ((=!! address-bits (!! 0)) (*set answer (+!! pvar-a pvar-b)))
         ((=!! address-bits (!! 1)) (*set answer (-!! pvar-a pvar-b)))
         ((=!! address-bits (!! 2)) (*set answer (*!! pvar-a pvar-b)))
         ((=!! address-bits (!! 3)) (*set answer (//!! pvar-a pvar-b))))
      answer))
```

This may now be used like any other !! function, as in:

```
(*set a (four-function!! (+!! a (!! 4)) (-!! a b)))
```

To pass a *LISP function as an argument, use *funcall. For example:

```
(defun *compose (*f *g x)
  (*funcall *f (*funcall *g x)))

(*set a (*compose 'sqrt!! '1+!! (!! 8)))
```

acts like:

```
(*set a (sqrt!! (1+!! (!! 8))))
```

See Section 5.5 for the precise definitions of both *defun and *funcall.

2.3.4 Communication

This section demonstrates how to cause the processors to communicate with one another.

One connectivity pattern that can be specified upon initialization is a two-dimensional grid in which each processor has a neighbor on the north, east, west, and south. It is possible to sum the value contained in a in the four neighbors of each processor, and store the result back in a as follows:

```
(*set a (+!! (pref-grid-relative!! a (!! -1) (!!  0))
             (pref-grid-relative!! a (!!  1) (!!  0))
             (pref-grid-relative!! a (!!  0) (!! -1))
             (pref-grid-relative!! a (!!  0) (!!  1))))
```

(Note that the effect of indexing out of bounds is ignored for the time being.)

To have the first 100 processors write the contents of field **b** into the field **a** of those processors whose addresses are 7 larger, do:

```
(*when (<!! (self-address!!) (!! 100))
  (setf (pref!! a (+!! (self-address!!) (!! 7))) b))
```

Finally, to find the maximum value of **a** in all even processors, one possibility is to do:

```
(*all
  (*when (=!! (!! 0) (mod!! (self-address!!) (!! 2)))
    (*max a)))
```

2.3.5 The Format of Function Definitions

The format of function definitions in this manual strive to be as compatible as possible with the format used in the COMMON LISP manual [Steele 84]. Argument names can restrict the type of an argument; argument names that end in the suffix "pvar" must be pvars. The name "integer-pvar" restricts the argument to a pvar whose fields in the currently selected set of processors must all contain integers.

2.4 Configuration Constants and Functions

*LISP makes it convenient to simulate in software a configuration of processors that is different from their physical configuration. It is important to write software that can take advantage of this flexibility. In addition, it is desirable to write software that will run on machines with differing amounts of physical hardware.

The variables defined in Section 2.4.1 specify the parameters of the machine as perceived by the user's program. If a program uses only these constants and functions, it is guaranteed to run in any configuration.

The size of simulated configuration of processors is specified through the `*cold-boot` function (see Chapter 8).

2.4.1 The Size of the Machine

`*number-of-processors-limit*`

This variable specifies the effective number of processors a user program sees. For a one-board machine with no virtual processors, this will be 512. For a complete machine with 65,536 physical processors, each simulating 16 processors, this constant will be 1,048,576.

`*log-number-of-processors-limit*`

This variable provides the logarithm, base 2, of the number of processors available.

`*number-of-dimensions*`

This variable is defined when `*cold-boot` is run. Its value is the number of dimensions given. `*Cold-boot` defaults this variable to be 2.

`dimension-size` *dimension*

This function returns one more than the maximum allowable grid address for the specified *dimension*. Note that *dimension* is zero based; for example, in a two dimensional machine, the first dimension is dimension zero and the second is dimension one. The number returned will be the same as that specified to the function `*cold-boot`.

2.4.2 Pre-defined Pvars

`t!!`

This is a pvar whose contents in each processor is the Lisp symbol `T`.

`nil!!`

This is a pvar whose contents in each processor is the Lisp symbol `NIL`.

Chapter 3

The Pvar Data Structure

The basic abstraction in *LISP is the pvar. A pvar is a Lisp object that references a field in the Connection Machine system. It contains everything necessary to describe the field. In ESSENTIAL *LISP, the contents of pvars may be any valid Lisp object. As in Lisp, coercion between data types and the amount of space necessary to accommodate an object in a pvar is automatically handled by *LISP.

This portion of the manual is organized as follows:

- Section 3.1 describes `*defvar`, `allocate!!`, and `*deallocate`, which create new pvars and destroy old ones.
- Section 3.2 describes some functions that return properties of a pvar.
- Section 3.3 discusses coercion of pvar types and lengths.
- Section 3.4 describes `*let` and `*let*`, which allocate temporary pvars.
- Section 3.5 describes `*set`, which allows each processor to set the value contained in the field defined by a pvar.
- Section 3.6 describes functions for explicitly putting Lisp values into the field defined by a pvar in a specific processor.

3.1 Constructing New Pvars

To create a relatively permanent, named pvar, use `*defvar` (analogous to the Lisp `defvar`). To create a permanent, unnamed pvar, use `allocate!!`; and to create a temporary, named pvar, use `*let` (analogous to the Lisp `let`).

`*defvar` *symbol* `&optional` *pvar documentation-string*

This creates a new pvar that is permanently allocated. *symbol* will contain the allocated pvar. The optional argument *pvar* may be any pvar or pvar expression. `*defvar` will create a new pvar of the type and size of the given *pvar*, initialize it to the given pvar's contents, and `setq` the *symbol* to that new pvar. If no *pvar* argument is given, the *symbol* will contain a pvar of value `nil!!`. Note that `*cold-boot` will reset the values of all pvars allocated by `*defvar`. This form returns *symbol*.

Some example uses of `*defvar` are:

```
(*defvar a)
(*defvar b (!! 5))
(*defvar c (+!! b (!! 6)))
(*defvar d t!!)
(*defvar e (self-address!!))
(*defvar f c)
```

`allocate!! &optional` *pvar*

This creates a relatively permanent pvar, just like `*defvar` except that the created pvar is simply returned. It is up to the user to store it someplace. `allocate!!` is very useful for making Lisp arrays or structures that contain permanent pvars. If no *pvar* is specified, then the returned pvar will have the value `nil!!`.

Some example uses are:

```
Assume c is a vector:

(dotimes (j (length c))
  (setf (aref c j) (allocate!!)))
```

and

```
(setq a (allocate!!))
(setq b (allocate!! (!! 5)))
(setf (aref c 4) (allocate!! t!!))
```

`*deallocate` *pvar*

This deallocates the given pvar if was permanently allocated (i.e., it was defined using either `allocate!!` or `*defvar`). It is an error to use a pvar after it has been deallocated. This function returns no values.

3.2 Determining the Type of a Pvar

`pvarp` *object*

This returns `T` if the argument is a pvar and `NIL` if it is not.

`type-of!!` *pvar*

This returns a pvar containing the COMMON LISP type of the argument *pvar*.

3.3 Coercion of Pvar Types and Lengths

When there is a need to use two pvars together, there is a potential need for *coercion*. This denotes the process whereby a pvar is converted to a pvar of a different size or type. Coercion may have to occur when one tries to set the contents of one pvar to the contents of another, when one tries to compare the contents of two pvars, or when one tries to do a numerical or logical operation on the contents of two pvars.

3.4 Allocating Local Pvars

*LISP maintains a stack of temporary pvars for its own purposes. When a `!!` function returns a pvar, it has been allocated on this stack. *LISP's ability to arbitrarily nest `!!` expressions stems from its maintenance of this stack. While this automatic allocation takes care of many situations, there are times when it is desirable to explicitly allocate a temporary variable. The ESSENTIAL *LISP functions for doing this are `*let` and `*let*`:

`*let` *({(symbol* `&optional` *pvar)}**) `&rest` *body*

The first expression following the `*let` should be a list of lists, each specifying one temporary variable. The elements of each sublist should consist of a Lisp symbol whose value will be the temporary pvar followed by an optional pvar that will be copied into the new one. This format is very similar to that of `*defvar`, the only difference being that several pvars can be created at once; these pvars only survive for the extent of the form. It is an error to try to refer to these pvars outside of the body of the `*let`. In other words, the *symbols* have lexical scope (as in COMMON LISP), whereas the pvars themselves have dynamic extent that terminates when the `*let` form is exited. (For a more detailed discussion of this, see Chapter 9 on page 37 of this manual.) `*let` returns the value of the last form of the body, regardless of whether that value is a pvar. It is legitimate to return a temporarily bound pvar.

`*let*` *({(symbol* `&optional` *pvar)}**) `&rest` *body*

This function behaves in the same manner as `*let` except that, as in COMMON LISP, the defining expressions are evaluated in parallel.

Example `*let` expressions might look like:

```
(*let* ((a)
       (b (!! 8))
       (c (*!! b (!! 528)))
       (d (!! -2.715))
       (e (self-address!!)))
   (some-pvar-function a b c d e)  ;This may modify a,b,c,d and/or e
   (+!! a b c d e))                ;This returns a pvar

;; take the global maximum of bits 16-31 of the self pointer
(*let ((a (load-byte!! (self-address!!) (!! 16) (!! 16))))
   (*max a))
```

3.5 Setting the Value of Pvars

The ***set** special form allows the contents of one pvar to be set to the contents of another. The field is set in those processors that are currently selected. A ***set** expression returns no value. ***set** takes multiple pairs, which are set sequentially. ***set** is often used in conjunction with ***all** to set the contents of one pvar to the contents of another in *all* processors.

***set** {*pvar-1 pvar-2*}*

This sets the contents of *pvar-1* to the contents of *pvar-2* in all processors of the currently selected set. Note that both of the arguments are evaluated.

Some examples of the use of this function are:

```
(*set a (+!! b c))

(*all (*set a (!! -1) b a c (!! -3)))

(*when (>!! d (!! 4)) (*set a b))
```

3.6 Reading and Writing Fields in Specific Processors

This section describes functions for getting data into and out of the Connection Machine processors. They are independent of the currently selected set and return, as Lisp values, the read or written Lisp value.

pref *pvar address*

This function returns, as a Lisp value, the contents of the field specified by *pvar* in the processor whose cube address is *address*. **setf** may be used with **pref** to write a value into a single processor of a pvar.

```
(pref foo 17)
```

returns the contents of pvar **foo** from processor 17.

```
(setf (pvar foo 17) (* 19 89))
```

sets the contents of pvar **foo** for processor 17 to 1691.

pref-grid *pvar* **&rest** *addresses*

This function returns, as a Lisp value, the contents of the field specified by *pvar* in the processor whose grid address is given by *addresses*. There must be as many addresses as there are dimensions (as specified with *cold-boot). **setf** may be used with **pref-grid** to write a value into a single processor of a pvar.

```
(pref-grid bar 4 7)
```

returns the contents of pvar **bar** from processor (4,7) (on a two-dimensionally configured machine).

```
(setf (pref-grid bar 4 7 8) (* 19 89))
```

sets the contents of pvar **bar** for processor (4,7,8) (on a three-dimensionally configured machine) to 1691.

Chapter 4

Processor Selection

Most operations are only executed in a subset of Connection Machine processors known as the *currently selected set*. Some of the special forms in ESSENTIAL *LISP that change the currently selected set are **`*all`**, **`*when`**, **`*cond`**, and **`*if`**. These special forms select processors based on the result of a pvar expression. Any processor in which the pvar expression evaluates to NIL is eliminated from the selected set.

Every user-defined function is expected to leave the currently selected set in the same state it was in when it was called. If one uses only the functions defined below, this discipline is automatically enforced. It is common for user functions to have a **`*all`** surrounding their bodies to ensure that they are starting out with the complete machine selected. Using the functions described in this section, the selected set is whittled down to select only the processors that should do a given operation. Note that the body of these forms is *always* executed, even if there are no selected processors.

`*all` **`&rest`** *body*

This form selects all processors. Its body is executed with the currently selected set equal to the entire machine. The value of the final expression in the body is returned whether it is a Lisp value or a pvar.

`*when` *pvar* **`&rest`** *body*

This form *subselects* from the currently selected set. Thus every processor that is unselected when **`*when`** is called remains unselected in the body of the **`*when`**. It selects processors in which *pvar* is non-NIL. **Even if there are no selected processors, ALL consequent forms are evaluated.** The value of the final expression in the body is returned whether it is a Lisp value or a pvar.

`*cond` *{(pvar {form}*)}**

This form is analogous to the Lisp cond. Unlike the Lisp cond, **`*cond`** evaluates all clauses; however, the currently selected set is determined by the *pvar* expressions. The nth consequent is evaluated with a selected set made up of initially selected processors that didn't pass the first $n-1$ tests, but did pass the nth one. `t!!` selects all remaining processors in the initial selected set. Even if there are no selected processors, all consequent forms are evaluated. Unlike Lisp's cond, this function returns no values and is executed only for its side effects (see also the description of `cond!!` in Section 5.4).

`*if` *pvar then-form* `&optional` *else-form*

This form is analogous to the Lisp `if`. *then-form* is performed in all processors of the currently selected set in which *pvar* is not `NIL`. The optional *else-form* form is evaluated in all other processors of the currently selected set in which the *pvar* is `NIL`. Even if there are no selected processors, both *then-form* and *else-form* are evaluated. Unlike Lisp's `if`, this function returns no values and is executed only for its side effects (see also the description of `if!!` in Section 5.4).

`with-css-saved` {*form*}*

This form is used whenever control flow would abnormally pass out of a *LISP form that restricts the currently selected set (e.g. using throw, return-from, or go, to leave the body of a **`*when`**). **`with-css-saved`** uses an unwind-protect to trap these events and force the currently selected set back to its state at the time the **`with-css-saved`** form was begun. **`with-css-saved`** returns what is returned by the evaluation of the last form of its body.

`do-for-selected-processors` *(symbol)* `&rest` *body*

This form evaluates *body* as many times as there are active processors, each time with *symbol* bound to the cube address of a different active processor. Like COMMON LISP's **`dotimes`** the *return* statement may be used to exit the **`do-for-selected-processors`** form immediately. Normally, **`do-for-selected-processors`** returns NIL.

Some examples of the use of these functions are:

```
(*all (*set a b))

(*when (=!! a b) (*set e (+!! c d)))

(*cond ((=!! a (!! 1)) (*set e (+!! b c)))
       ((not!! (=!! c d)) (*set f (*!! b c)))
       (t!! (*set f (!! 9))))

(*if (=!! c d) (*set e f) (*set g h))

(*set a (=!! b c))
(*when a (*set b (-!! b))
```

and

```
(*defun f (x y)
  "Returns y divided by x for y greater than 0.  Returns NIL
   if any x is 0.
  "
  (block foo
(with-css-saved
  (*when (>!! (!! 0) y)
    (*if (=!! (!! 0) x)
       (return-from foo nil)
       (/!! y x)
      )))))
```

Chapter 5

!! Functions

This section introduces a variety of functions that work within an individual processor and that return pvars as values. Recall that in *LISP, it is conventional for functions that return pvars to end in !!.

This portion of the manual describes the following:

- the predicates that return Boolean pvars (Section 5.1)
- the logical functions (Sections 5.2.1 and 5.2.2)
- the numerical functions (Section 5.3)
- the forms that allow the user to define his or her own !! functions (Section 5.4)

5.1 Predicate !! Functions

The functions in this section are typically used as predicates within *when expressions. They return a pvar that contains T in all processors of the currently selected set in which the predicate holds, and a NIL in those in which it does not. In comparing pvars of different types, the less expensive pvar types are coerced to the more expensive types (e.g., integers are coerced into floats before comparision is done).

=!! *numeric-pvar* &rest *numeric-pvars*

This returns a pvar that contains T in each processor where the argument pvars contain equal values after type coercion and NIL elsewhere. To see if a pvar is equal to a Lisp constant, use an expression like:

```
(=!! foo (!! 5))
```

If only one argument pvar is given, the returned pvar will be t!!.

/=!! *numeric-pvar* &rest *numeric-pvars*

This returns a pvar that contains T in each processor where the argument pvars contain unequal values after type coercion and NIL elsewhere. If only one argument is given, the returned pvar will be t!!.

(NOTE: On Symbolics Lisp machines, when using the Zetalisp reader, the symbol "/" is a reader macro character, so one must use "//=!!".)

`<!!` *numeric-pvar* `&rest` *numeric-pvars*

This returns a pvar that contains `T` in each processor where the argument pvars contain values which, after type coercion, are in strictly increasing order and `NIL` elsewhere. If only one argument pvar is given, the returned pvar will be `t!!`.

`>!!` *numeric-pvar* `&rest` *numeric-pvars*

This returns a pvar that contains `T` in each processor where the argument pvars contain values which, after type coercion, are in strictly decreasing order and `NIL` elsewhere. If only one argument pvar is given, the returned pvar will be `t!!`.

`<=!!` *numeric-pvar* `&rest` *numeric-pvars*

This returns a pvar that contains `T` in each processor where the argument pvars contain values which, after type coercion, are in non-decreasing order and `NIL` elsewhere. If only one argument pvar is given, the returned pvar will be `t!!`.

`>=!!` *numeric-pvar* `&rest` *numeric-pvars*

This returns a pvar that contains `T` in each processor where the argument pvars contain values which, after type coercion, are in non-increasing order and `NIL` elsewhere. If only one argument pvar is given, the returned pvar will be `t!!`.

5.2 Logical !! Functions

5.2.1 Logical !! Operations on Numbers

This section contains a variety of Boolean functions that operate bitwise on the bits of the fields described by the argument pvars and return a pvar that holds the result. These functions may be used only on pvars whose contents are integers. The returned pvar will usually contain positive integers.

`lognot!!` *integer-pvar*

This returns a pvar whose bits are the logical complement of the bits in *integer-pvar*.

`logior!!` `&rest` *integer-pvars*

This returns a pvar whose bits are the logical inclusive or of the bits in *integer-pvars*. If there are no *pvars*, then `(!! 0)` is returned.

`logxor!!` `&rest` *integer-pvars*

This is the parallel equivalent of the COMMON LISP function *logxor*.

logand!! &rest *integer-pvars*

This returns a pvar whose bits are the logical and of the bits in *integer-pvars*. If no *pvars* are given, then (!! -1) is returned.

logeqv!! &rest *integer-pvars*

This is the parallel equivalent of the COMMON LISP function *logeqv*.

5.2.2 Logical !! Operators

*LISP provides several logical operators. Some of these operators (and!! and or!!) are special because they temporarily subselect the currently selected set as they evaluate their arguments. The rest of the operators are normal !! functions.

As in COMMON LISP, a value is true if it is anything other than NIL.

and!! &rest *pvars*

This evaluates the *pvars* from left to right in all selected processors. As soon as one of the *pvars* evaluates to NIL in a processor, that processor is removed from the currently selected set for the remainder of the **and!!**. **and!!** will return the value of the last pvar for all selected processors in which all the *pvars* are true and NIL otherwise. If no *pvars* are given, then **t!!** is returned.

or!! &rest *pvars*

This evaluates the *pvars* from left to right in all selected processors. As soon as one of the *pvars* evaluates to non-NIL in a processor, that processor is removed from the currently selected set for the remainder of the **or!!**. The value returned for each processor will be the first pvar that evaluated to non-NIL. If none of the pvars are true, then NIL is returned. If no *pvars* are given, then **nil!!** is returned.

xor!! &rest *pvars*

This performs the xor function on all the *pvars*. If no *pvars* are given, then **nil!!** is returned. In each processor this returns T if there are an odd number of arguments that are true and otherwise returns NIL.

eql!! *pvar1 pvar2*

This is the parallel equivalent of the COMMON LISP function eql.

eq!! *pvar1 pvar2*

This is the parallel equivalent of the COMMON LISP function eq.

integerp!! *pvar*

This is the parallel equivalent of the COMMON LISP function **integerp**.

floatp!! *pvar*

This is the parallel equivalent of the COMMON LISP function **floatp**.

`numberp!!` *pvar*

This is the parallel equivalent of the COMMON LISP function `numberp`.

`not!!` *pvar*

This returns T for all processors in which *pvar* is `NIL`.

5.3 Numerical !! functions

This section describes the elementary numerical functions. They each return results of the same type as the most expensive of their arguments (e.g. if all arguments are integers, the result will generally be an integer; but if any arguments are floats, the result will be a float).

`!!` *lisp-expression*

This returns a pvar containing the result of *lisp-expression* in each processor.

`+!!` **`&rest`** *numeric-pvars*

This adds the contents of the argument pvars. The result has as many extra bits as are needed to hold the result. If there are no arguments, then `(!! 0)` is returned.

`-!!` *numeric-pvar* **`&rest`** *numeric-pvars*

This subtracts the contents of the second through last argument's pvars from the contents of the first. If there is only one argument, the result is that argument negated.

`*!!` **`&rest`** *numeric-pvars*

This multiplies the contents of the argument pvars. If there are no arguments, then `(!! 1)` is returned.

`/!!` *numeric-pvar* **`&rest`** *numeric-pvars*

This returns the quotient of *pvar* by the rest of the *pvars*. If there is only one argument, the result is the inverse of *pvar*. Note: `/!!` presently always returns a pvar whose contents are all floating point numbers. If there is only one argument, it is an error if that argument has any field whose value is 0. If there is more than one argument, it is an error if any argument but the first has any field whose value is 0.

`1+!!` *numeric-pvar*

This increments the argument pvar by 1.

`1-!!` *numeric-pvar*

This decrements the argument pvar by 1.

`min!!` *numeric-pvar* **`&rest`** *numeric-pvars*

This returns a pvar that is the minimum of all the argument pvars.

`max!!` *numeric-pvar* `&rest` *numeric-pvars*

This returns a pvar that is the maximum of all the argument pvars.

`mod!!` *numeric-pvar integer-pvar*

This function is identical in effect to the COMMON LISP mod function (except that it operates in parallel!). It is an error if any field of *integer-pvar* is 0.

`truncate!!` *numeric-pvar*

This returns *numeric-pvar* coerced into an integer by truncating any fractional part.

`float!!` *numeric-pvar*

This converts *numeric-pvar* to a floating point number.

`sqrt!!` *non-negative-pvar*

This returns the non-negative square root of the given pvar.

`random!!` *limit-pvar*

This returns a pvar whose contents is a random value between 0 inclusive and *limit-pvar* exclusive for each processor.

5.4 Miscellaneous !! Functions

`load-byte!!` *from-pvar position-pvar size-pvar*

This function returns a pvar whose contents are positive integers. It consists of bits extracted from *from-pvar* starting at bit position *position-pvar*, where 0 represents the least significant bit. In any processor in which zero bits are extracted, the resulting field contains zero. This operation is especially fast when both *position-pvar* and *size-pvar* are constants as in (!! *lisp-value*). *from-pvar* must be a pvar containing integers, while *position-pvar* and *size-pvar* must be pvars containing non-negative integers. Out of range bits are treated as zero for positive integers (for example, `(load-byte!! (!! 1) (!! 2) (!! 3))` returns a pvar that contains zero in each processor), and one for negative integers (for example, `(load-byte!! (!! -1) (!! 2) (!! 3))` returns a pvar that contains 7 in each processor).

`deposit-byte!!` *into-pvar position-pvar size-pvar byte-pvar*

This returns a pvar whose contents are a copy of *into-pvar* with the low order *size-pvar* bits of *byte-pvar* inserted into the bits starting at location *position-pvar*.

When the *into-pvar* is positive (negative), zeros (ones) are appended as high order bits of *byte-pvar* as needed. The returned value may have more bits than *into-pvar* if the inserted field extends beyond the most significant bit of *into-pvar*. For example, `(deposit-byte!! (!! 3) (!! 1) (!! 2) (!! 2))` will return `(!! 5)`. This function is especially fast when both *position-pvar* and *size-pvar* are constants, as in (!! *lisp-value*). *Into-pvar* and *byte-pvar* must contain integers, while *position-pvar* and and *size-pvar* must be pvars containing non-negative integers only.

if!! *pvar then-pvar else-pvar*

This returns a pvar that contains the contents of the *then-pvar* in all processors in which *pvar* is non-NIL, and the contents of *else-pvar* in all processors in which *pvar* is NIL.

This is roughly equivalent to:

```
(*let ((result)
       (temp-pred pvar))
  (*when temp-pred
     (*set result then-pvar))
  (*when (not!! temp-pred)
     (*set result else-pvar))
  result
 )
```

An example that demonstrates the usefulness of **if!!** is the following function to take the absolute value:

```
(*defun abs!! (pvar)
  (if!! (>!! pvar (!! 0)) pvar (-!! pvar)))
```

cond!! *{(pvar {form}*)}**

If there are no clauses, **cond!!** returns **nil!!**. Otherwise, if there is more than one clause, **cond!!** is roughly equivalent to the following pseudo-code:

```
(if!! pvar-1
      (progn all-the-forms-for-clause1)
      (cond!! (rest clauses))
```

However, if there are no forms for a given clause, the *pvar* itself is used as the value of the clause, analogous to the COMMON LISP **cond**.

enumerate!!

This returns a pvar that contains a unique number in each selected processor from 0 up to one less than the number of selected processors.

5.5 User !! and * Functions

This section provides information necessary for defining and using new !! functions.

Pvar arguments are passed by reference, not by value. Thus the contents of pvars passed as arguments can be changed using ***set**. It is generally considered poor form for a function to modify one of its arguments; instead, most *LISP functions return a new pvar whose contents may be a modified copy of one of the arguments.

`*defun`

This is analogous to the COMMON LISP **defun** and must be used in place of it in defining all user functions that might take as an argument a pvar or that might return a pvar as a result. This returns, as a symbol, the name of the function being defined. Like the COMMON LISP **defun**, the body may contain declaration and a documentation string.

`*funcall` *function* **`&rest`** *arguments*

This is used just like COMMON LISP's **funcall**, but in functions defined with **`*defun`**.

`*apply` *function arg* **`&optional`** *more-args*

This is used just like COMMON LISP's **apply**, but in functions defined with **`*defun`**.

Chapter 6

Parallel Global Memory References

6.1 Parallel Global Memory References

This section describes the mechanisms for moving data between the Connection Machine processors in parallel. The high-speed Connection Machine router network provides global memory references from many processors in parallel. The function that does communication is `pref!!`, the parallel version of `pref`.

`pref!!` *pvar-expression cube-address-pvar*

`pref!!` will return a pvar that contains the value of *pvar-expression* from the processors addressed by *cube-address-pvar*. This function evaluates *pvar-expression* differently from other *LISP operators; instead of evaluating the *pvar-expression* in the currently selected set, it is evaluated **in the context of the processor from which the data is being retrieved.** Unlike the *pvar-expression*, *cube-address-pvar* is evaluated normally (i.e., in the processors of the currently selected set).

If the value of *pvar-expression* in a single processor is being accessed by more than one other processor, the Connection Machine system arranges for all those other processors to get the same value.

`pref-grid!!` *pvar-expression* `&rest` *pvar-addresses* `&key border-pvar`

`pref-grid!!` will return a pvar that contains the value of *pvar-expression* from the processors addressed by *pvar-addresses*. This function evaluates *pvar-expression* differently from other *LISP operators; instead of evaluating the *pvar-expression* in the currently selected set, it is evaluated **in the context of the processor from which the data is being retrieved.**

There must be as many *pvar-addresses* as there are Connection Machine dimensions. Unlike the *pvar-expression*, the *pvar-addresses* are evaluated like normal expressions (i.e., in the processors of the currently selected set).

It is an error to read from a non-existent processor. However, if *border-pvar* is provided, and if the *pvar-addresses* in a given processor p access a non-existent processor, then the value stored in *border-pvar* in processor p is returned instead.

Again, if the value of *pvar-expression* in a single processor is being accessed by more than one other processor, the Connection Machine system arranges for all those other processors to get the same value.

pref-grid-relative!! *pvar-expression* **&rest** *relative-pvar-addresses* **&key** **border-pvar**

This function behaves like **pref-grid!!**, except that relative addressing is used instead of absolute addressing. An example of the use of this function is given later in this chapter.

Like the serial function **pref**, **setf** may be applied to **pref!!** and **pref-grid!!** to write into memory instead of reading from it. In this case, the *pvar-expression* must be a symbol because one may not store a value into an expression; this symbol will be referred to as *dest-pvar*.

When using **setf** in this manner, *dest-pvar* is modified only in those processors that were accessed. Processors that were not written into will retain the previous contents of *dest-pvar*. An error is signalled if a non-existent processor is addressed. This occurs when an address is out of the bounds specified by the current Connection Machine configuration.

Although the Connection Machine hardware is capable of accessing the same memory for several readers without problems, the user must instruct it on how to handle collisions when several processors are simultaneously writing to the same location. Should a processor be written into by several other processors in a single memory reference, **pref!!** and its relatives (in combination with **setf**) will signal an error. The function ***pset** allows multiple writes to combine in various ways without producing errors.

***pset** *combinator value-pvar dest-pvar cube-address-pvar*

For all selected processors, *value-pvar* will be written into *dest-pvar* of the processor addressed by *cube-address-pvar*. When more than one value is written into the same address, the *combinator* determines how the values are combined. *combinator* may be one of the following:

1. **:default** — If the same address is written twice, an error is signalled. This is the same as using **setf** with **pref!!**.

2. **:overwrite** — Only one write per address is successful. All other writes are discarded.

3. **:or** — If two or more values are written into a single processor, the final value will be the logical OR of those values.

4. **:and** — If two or more values are written into a single processor, the final value will be the logical AND of those values.

5. **:logior** — If two or more values are written into a single processor, the final value will be the bitwise OR of those values.

6. **:logand** — If two or more values are written into a single processor, the final value will be the bitwise AND of those values.

7. **:add** — If two or more values are written into a single processor, the final value will be the numerical SUM of those values.

8. **:max** — If two or more values are written into a single processor, the final value will be the numerical MAXIMUM of those values.

9. `:min` — If two or more values are written into a single processor, the final value will be the numerical MINIMUM of those values.

***pset-grid** *combinator value-pvar dest-pvar* **&rest** *grid-address-pvars*

This is analogous to ***pset**, except that the grid addressing is used.

Here are some sample uses of **pref!!**:

```
(*set a (pref!! b (!! 100)))
```

This reads the contents of b from processor 100 and stores it in pvar a. Only those components of a which are in processors in the currently selected set are modified.

These two forms are equivalent:

```
(*all (setf (pref!! b (self-address!!)) a))

(*all (*set b a))
```

This example writes pvar a into pvar b of all processors. Processors can read from themselves just as easily as they can read from other processors.

These two forms are equivalent:

```
(*all
  (*when (>!! (self-address!!) (!! 0))
     (*set a (pref!! (self-address!!)
                     (1-!! (self-address!!))))))

(*all
  (*when (>!! (self-address!!) (!! 0))
     (*set a (1-!! (self-address!!)))))
```

Note that this example demonstrates that the *pvar-expression* of **pref!!** is evaluated in the processor from which the data is being fetched. Remember that it is an error to read or write from a non-existent processor. In the above examples, the form `(*when (>!! (self-address!!) (!! 0))` prevents that from happening.

This function:

```
(*defun sum-a-pvar (pvar)
  (pref
    (*let ((the-sum-goes-here))
      (*all (*pset :add pvar the-sum-goes-here) (!! 47)))
      the-sum-goes-here)
    47)
```

returns the sum of a pvar over all the Connection Machine processors. (Processor 47 was chosen to contain the sum for demonstration purposes only.)

The following is an example of **pref-grid-relative!!**:

```
(*all
  (*set color
        (/!!
          (+!!
            (pref-grid-relative!! color (!! -1) (!! 0) :border-pvar (!! 1))
            (pref-grid-relative!! color (!! 0) (!! -1) :border-pvar (!! 1))
            (pref-grid-relative!! color (!! 0) (!!  1) :border-pvar (!! 1))
            (pref-grid-relative!! color (!! 1) (!!  0) :border-pvar (!! 1))
            color)
          (!! 5))))
```

This example causes the value of the **color** pvar in each processor to be averaged with the 4 processors to its north, east, west and south.

6.2 Processor Addressing

The following are functions that deal with address generation and translation:

self-address!!

This function returns a pvar that contains the cube address of each selected processor.

self-address-grid!! *dimension-pvar*

This returns a pvar that contains the grid address, in the specified dimension, of each selected processor. Each processor may specify a different dimension through *dimension-pvar.*

grid-from-cube-address *cube-address dimension*

This function takes a *cube-address* and returns the grid address for the specified *dimension.* This function executes entirely in the host computer.

`cube-from-grid-address` *address-pvar* **`&rest`** *address-pvars*

This function translates a grid address consisting of (possibly) several *address-pvars* into a cube address. This function executes entirely in the host computer.

`grid-from-cube-address!!` *cube-address-pvar dimension-pvar*

This function takes a *cube-address-pvar* and returns a pvar containing the grid address for the specified *dimension-pvar* for each selected processor.

`cube-from-grid-address!!` *address-pvar* **`&rest`** *address-pvars*

This function translates a grid address consisting of (possibly) several *address-pvars* into a cube address for each selected processor.

Chapter 7

Global Operations

The following functions collect data from many processors in the Connection Machine computer:

`*logior` *integer-pvar*
This returns a Lisp value that is the bitwise logior of the contents of *integer-pvar* in all selected processors. This returns the Lisp value 0 if there are no selected processors.

`*logand` *integer-pvar*
This returns a Lisp value that is the bitwise logand of the contents of *integer-pvar* in all selected processors. This returns the Lisp value -1 if there are no selected processors.

`*min` *numeric-pvar*
This returns a Lisp value that is the minimum of the contents of *numeric-pvar* in all selected processors. This returns the Lisp value `NIL` if there are no selected processors.

`*max` *numeric-pvar*
This returns a Lisp value that is the maximum of the contents of *numeric-pvar* in all selected processors. This returns the Lisp value `NIL` if there are no selected processors.

`*or` *pvar*
This returns a Lisp value of `T` if the contents of *pvar* is non-`NIL` in any selected processor; otherwise, it returns `NIL`. If there are no selected processors, this function also returns `NIL`. For example, to determine if there are any processors currently selected, use `(*or t!!)`, which returns `T` only if there are selected processors.

`*and` *pvar*
This returns a Lisp value of `T` if the contents of *pvar* is non-`NIL` in every selected processor; otherwise, it returns `NIL`. If there are no selected processors, this function returns `T`.

`*sum` *numeric-pvar*
This returns a Lisp value that is the sum of *numeric-pvar* in every selected processor. This returns the Lisp value 0 if there are no selected processors.

`*product` *numeric-pvar*
This returns a Lisp value that is the product of *numeric-pvar* in every selected processor. This returns the Lisp value 1 if there are no selected processors.

Chapter 8

Actually Using the Hardware

This section describes the two *LISP functions (*cold-boot and *warm-boot) that allow the user to actually use the hardware.

***cold-boot &key initial-dimensions**

This function initializes *LISP and must be called immediately after loading in the *LISP software. It resets the internal state of the *LISP system, as well as the Connection Machine hardware. All *defvar pvars are reallocated and their initial values are recomputed according to the order in which they were defined.

In addition, the user may specify the *initial-dimensions* of the machine. This argument is a list of dimension sizes. This affects the behavior of *LISP functions such as pref-grid!! and pref-grid-relative!!. The dimensions must be powers of 2. If no *initial-dimensions* are specified, then it defaults to the same values as in the previous call to *cold-boot.

*LISP will try to attach the necessary amount of Connection Machine hardware to satisfy the user requirements. If the hardware is not large enough, or is not of the proper shape, *LISP will try running virtual processors. The only difference the user will ever notice is a degradation of performance. An error will be signalled if sufficient hardware is not available.

*cold-boot is typically called by the initialization function of the user's software. Under normal circumstances, this need only be called at the start of a session.

Here are some typical calls:

```
;; allocate 16K processors
(*cold-boot :initial-dimensions (list (expt 2 14)))

;; use same initializations as previous call
(*cold-boot)

;; 2-D grid (128 x 128)
(*cold-boot :initial-dimensions (list 128 128))

;; allocate a 16 dimension hypercube
(*cold-boot :initial-dimensions (make-list 16 :initial-element 2))
```

`*warm-boot`

This function must be called whenever a *LISP program is abnormally terminated for any reason. The function will reset only certain internal *LISP and Connection Machine hardware states.

It is wise to call this function at the beginning of major entry points in the user's software, since previously run code may have left the Connection Machine hardware in an inconsistent state.

Chapter 9

Potentially Troublesome Situations

This chapter describes potential ambiguities and trouble spots in the ESSENTIAL *LISP language definition.

9.1 Pvar Values in Non-selected Processors

It is an error to depend on the value of a pvar in a processor which was not in the currently selected set at the time the pvar was created.

For instance, in the following:

```
(*when (<!! (self-address!!) (!! 10))
  (*let ((foo (self-address!!)))
    (print (pref foo 20))
   ))
```

the ESSENTIAL *LISP language definition does not define the value printed in the above example.

9.2 The Extent of Pvars

Unlike COMMON LISP, pvars defined using `*let` or `*let*` have dynamic extent – that is, it is an error to reference the value of a pvar once the body of its defining `*let` or `*let*` has been exited.

For example:

```
(*defun will-not-work (pvar constant)
  (funcall
    (*let ((xyzzy (!! constant))
      #'(lambda (x) (*sum (+!! (!! x) xyzzy))
     )
    (pref pvar 0)
   ))
```

Since the body of the `*let` defining xyzzy has been exited at the time the lambda-defined function is actually called, the ESSENTIAL *LISP language definition makes no guarantee that the pvar xyzzy will still contain `constant` anywhere.

9.3 Using Mapcar on Functions that Return Pvars

Consider the following code:

```
(let ((pvar-list nil))
  (setq pvar-list (mapcar #'!! '(1 2 3)))
  (*set pre-defined-pvar-1
(+!! (first pvar-list) (second pvar-list)))
  (*set pre-defined-pvar-2
(+!! (first pvar-list) (second pvar-list)))
 )
```

`pvar-list` is a list of pvars that have been allocated on the ESSENTIAL *LISP stack. The ESSENTIAL *LISP language definition makes no guarantees as to how long these pvars will remain inviolate, since they are on the stack. There is absolutely no guarantee that `pre-defined-pvar-1` and `pre-defined-pvar-2` will contain the same values.

9.4 *Warm-boot

Whenever an ESSENTIAL *LISP program has an error, and the user aborts back to top level, the `*warm-boot` function must be called before attempting to run any ESSENTIAL *LISP code again. This is because the *currently selected set* of processors is not reset when COMMON LISP aborts back to top level, and `*warm-boot` resets the *currently selected set* to be all processors. Very bizarre behavior results when this dictate is not followed. The function `display-active-processors` can be used to determine the current state of the *currently selected set.*

Appendix A

Some Example Programs

```
;;; -*- SYNTAX: common-lisp; MODE: lisp; BASE: 10; PACKAGE: *LISP-CL; MUSER: YES; Fonts:
CPTFONTB,CPTFONT -*-


;;; Cliff Lasser  2/86

;;;
;;;  This file contains working examples for the Essential *Lisp manual.
;;;  Although they are in Common Lisp, they do use a couple of ZetaLisp
;;;  features: the LOOP macro and 3600 specific terminal control.
;;;



;;;
;;;  Example # 1:   Conway's life
;;;



;;;
;;;  Conway's Life is a simple 2-dimensional cellular automata algorithm.
;;;  Each cell in the 2-d grid is either alive or dead.  A cell becomes alive
;;;  if exactly three of its neighbors are alive.  A cell stays alive if two
;;;  or three of its neighbors are alive.  In all other cases a cell becomes
;;;  dead or remains dead.  A cell's neighbors are the cells to its east,
;;;  west, north and south, and to its NW, NE, SW and SE.
;;;
```

```
;;;
;;; This is the main entry point for the Life program.  It will do all
;;; initialization, then a few cycles of Life.
;;;
(DEFUN DO-LIFE ()
  (INITIALIZE-LIFE)
  (DISPLAY-LIFE-GRID)
  (LIFE-CYCLE 100))                        ;run 100 cycles of life


;;;
;;; This declares a PVAR that exists in all processors.  Each cell in the
;;; grid is either alive or dead.  If the cell is alive, CELL-ALIVE-P will
;;; contain T.  If it is dead, it will contain NIL.
;;;
(*DEFVAR CELL-ALIVE-P)


;;;
;;; This will initialize *Lisp and create a random pattern on the Life grid.
;;; By default, 30% of the cells will be alive
;;;
(DEFUN INITIALIZE-LIFE (&OPTIONAL (PERCENT-ON 30))
  ;; Initialize the Connection Machine system (CMS) and *Lisp.  Remember
  ;; to always call this before actually doing anything on the CMS.
  (*COLD-BOOT)

  ;; now fill the machine with a random pattern
  (*SET CELL-ALIVE-P (<!! (RANDOM!! (!! 100)) (!! PERCENT-ON)))
  )


;;;
;;; This will display the Life grid on the user's terminal
;;;
(DEFUN DISPLAY-LIFE-GRID ()
  ;; clear the screen (Symbolics 3600 specific code)
  (ZL:SEND TV:SELECTED-WINDOW :CLEAR-WINDOW)
  ;; step through each location on the x-y grid
  (LOOP FOR X FROM 0 BELOW (DIMENSION-SIZE 0)
        DO
    ;; go to next line on screen.
    (FORMAT T "~%")
    (LOOP FOR Y FROM 0 BELOW (DIMENSION-SIZE 1)
          DO
      ;; if the cell is alive, print a *.  Otherwise, a space
      (IF (PREF-GRID CELL-ALIVE-P X Y)
          (PRINC "*")
          (PRINC " ")))))

;;;
;;; This function will repeatedly run the Life algorithm and display the life
;;; grid.
;;;
(DEFUN LIFE-CYCLE (NUMBER-OF-CYCLES)
  (LOOP FOR CYCLE FROM 0 BELOW NUMBER-OF-CYCLES
        DO
    ;; allocate some temporary storage for counting the number of alive
    ;; neighbors for each cell in the grid.
    (*LET ((NUMBER-OF-ALIVE-NEIGHBORS (!! 0)))

      ;; each cell totals up the number of alive neighbors.  Be careful to
      ;; not include the cell itself in the total.
      (LOOP FOR X FROM -1 TO 1
            DO
        (LOOP FOR Y FROM -1 TO 1
              DO
          (WHEN (NOT (= 0 X Y))
            (*WHEN (PREF-GRID-RELATIVE!! CELL-ALIVE-P (!! X) (!! Y))
              ;; only those cells with the specific neighbor alive get to
              ;; increment the count.
              (*SET NUMBER-OF-ALIVE-NEIGHBORS
                    (+!! NUMBER-OF-ALIVE-NEIGHBORS (!! 1)))))))

      ;; based on the count of alive neighbors, each cell decides whether to
      ;; become alive or dead.
      (*SET CELL-ALIVE-P (OR!! (AND!! CELL-ALIVE-P
                                      (=!! (!! 2) NUMBER-OF-ALIVE-NEIGHBORS))
                               (=!! (!! 3) NUMBER-OF-ALIVE-NEIGHBORS))))
    ;; display the life grid.
    (DISPLAY-LIFE-GRID)
    ))
```

```
;;;
;;;  Example # 2:  Shortest path in a graph
;;;


;;;
;;; In this example, we have the CMS find the shortest path between two cities
;;; in a graph of connected cities.  Each connection between cities contains
;;; the distance between the two connected cities.
;;;
;;;
;;; The data structure used is very simple:  We split up the Connection
;;; Machine processors into two sets.  The first set represents the cities
;;; themselves.  The second set represents the connections between cities.
;;;


(*DEFVAR CITY-P NIL       "true when this processor contains a city")
(*DEFVAR CONNECTION-P NIL "true when this processor contains a connection item")

(*DEFVAR CITY-NAMES NIL "This contains the name of city for each city processor")
(*DEFVAR CITY-DISTANCE-FROM-START NIL "Distance of a city from the START processor")

(*DEFVAR CONNECTED-CITY-FROM NIL "One of the cities in a connection")
(*DEFVAR CONNECTED-CITY-TO   NIL "The other city in a connection")
(*DEFVAR CONNECTION-DISTANCE NIL "The distance between the connected cities")




;;;
;;; This defines the graph of connected cities
;;;
(DEFPARAMETER LIST-OF-ALL-CITIES '(NEW-YORK LOS-ANGELES BOSTON
                                   WASHINGTON SAN-FRANCISCO MIAMI CHICAGO))

(DEFPARAMETER CITY-CONNECTIONS-LIST
              '((NEW-YORK (BOSTON 220) (WASHINGTON 500))
                (LOS-ANGELES (SAN-FRANCISCO 600) (WASHINGTON 2500))
                (BOSTON (NEW-YORK 220) (WASHINGTON 600) (CHICAGO 1000))
                (WASHINGTON (NEW-YORK 500) (BOSTON 600) (LOS-ANGELES 2500))
                (SAN-FRANCISCO (CHICAGO 1500) (LOS-ANGELES 600))
                (MIAMI (CHICAGO 2500))
                (CHICAGO (BOSTON 1000) (SAN-FRANCISCO 1500) (MIAMI 2500))))
```

```
;;;
;;; This is a hash table that contains the processor number assigned to each
;;; of the cities.
;;;
(DEFVAR PROCESSORS-FOR-CITIES (MAKE-HASH-TABLE))

;;;
;;; Here is the top level function
;;;
(DEFUN DO-FIND-SHORTEST-PATH (START-CITY STOP-CITY)
  (BUILD-GRAPH)
  (FIND-SHORTEST-PATH START-CITY)
  (PRINT-SHORTEST-PATH STOP-CITY))


(DEFVAR *NEXT-FREE-PROCESSOR* NIL "This is used to allocate processors")

;;;
;;; This function will load the graph into the CMS
;;;
(DEFUN BUILD-GRAPH ()
  ;; allocate more processors than we will need
  (*COLD-BOOT :INITIAL-DIMENSIONS '(100))

  ;; initialize all the processors in the machine to be neither connections
  ;; nor cities
  (*SET CITY-P NIL!!)
  (*SET CONNECTION-P NIL!!)

  ;; start allocating processors with processor 0
  (SETQ *NEXT-FREE-PROCESSOR* 0)

  ;; start by assigning processors to cities.  All we have to do is write a T
  ;; into CITY-P, and write the name of the city into CITY-NAMES (this is
  ;; just a convenience)
  (LOOP FOR CITY IN LIST-OF-ALL-CITIES
        FOR CITY-PROCESSOR = *NEXT-FREE-PROCESSOR*
        DO
    (SETF (GETHASH CITY PROCESSORS-FOR-CITIES) CITY-PROCESSOR)
    (SETF (PREF CITY-P     CITY-PROCESSOR) T)
    (SETF (PREF CITY-NAMES CITY-PROCESSOR) CITY)

    (INCF *NEXT-FREE-PROCESSOR*))
  ;; loop through all the cities, and set up their connections.  This means
  ;; writing a T into CONNECTION-P, as well as the cities at both ends of the
  ;; connection, and the distance between them.
  (LOOP FOR ITEM IN CITY-CONNECTIONS-LIST
        FOR CITY =        (FIRST ITEM)
        FOR CONNECTIONS = (REST  ITEM)
        DO
    (LOOP FOR CONNECTION IN CONNECTIONS
          FOR CONNECTED-CITY = (FIRST  CONNECTION)
          FOR DISTANCE       = (SECOND CONNECTION)
          FOR CONNECTION-PROCESSOR = *NEXT-FREE-PROCESSOR*
          DO
      (SETF (PREF CONNECTION-P        CONNECTION-PROCESSOR) T)
      (SETF (PREF CONNECTION-DISTANCE CONNECTION-PROCESSOR) DISTANCE)

      (SETF (PREF CONNECTED-CITY-FROM CONNECTION-PROCESSOR)
            (GETHASH CITY           PROCESSORS-FOR-CITIES))
      (SETF (PREF CONNECTED-CITY-TO   CONNECTION-PROCESSOR)
            (GETHASH CONNECTED-CITY PROCESSORS-FOR-CITIES))

      (INCF *NEXT-FREE-PROCESSOR*)
      ))
  ;; End of building the graph routine
  )
;;;
;;;Here is the actual algorithm for computing the length of the shortest path
;;;between two cities:
;;;
;;;We define one of the two cities as the START city and the other as the
;;;STOP city.
;;;
;;;(1) All cities set their distance from the START city to some very large
;;;    number. The START city sets its distance to zero.
;;;
;;;(2) All connections fetch the distance pvar of their CONNECTED-CITY-FROM, and
;;;    add on their CONNECTION-DISTANCE.
;;;
;;;(3) All connections send the result of the previous set to their
;;;    CONNECTED-CITY-TO.  The send is done with a :MIN combiner.
;;;
;;;(4) All cities set their distance from the START city to the minimum of their
;;;    current distance and the value sent by the previous step.
;;;
```

```
;;;(5) If any city got a newer mimimum distance, then go cycle back to step
;;;    (2) again.


(DEFUN FIND-SHORTEST-PATH (START-CITY)
  ;; translate the start city into a processor number
  (SETQ START-CITY (GETHASH START-CITY PROCESSORS-FOR-CITIES))

  ;; (1) All cities set their distance from the START to some very large
  ;; number.  The START city sets its distance to zero.

  (*WHEN CITY-P (*SET CITY-DISTANCE-FROM-START (!! 30000)))
  (SETF (PREF CITY-DISTANCE-FROM-START START-CITY) 0)

  ;; allocate some storage for the computation of the next CITY-DISTANCE-FROM-START
  (*LET ((NEW-CITY-DISTANCE-FROM-START CITY-DISTANCE-FROM-START))

    (LOOP WITH ANY-NEW-DISTANCE-SHORTER-P ;This is T when we need to loop again.
          DO                              ;This will loop until the WHILE ...
                                          ;... below is false

      ;;(2) All connections fetch the distance pvar of their CONNECTED-CITY-FROM, and
      ;;    add on their CONNECTION-DISTANCE.

      (*WHEN CONNECTION-P
        (*LET ((DISTANCE-OF-CONNECTED-CITY
                 (+!! CONNECTION-DISTANCE (PREF!! CITY-DISTANCE-FROM-START
                                                  CONNECTED-CITY-FROM))))

          ;;(3) All connections send the result of the previous set to their
          ;;    CONNECTED-CITY-TO.  The send is done with a :MIN combiner.

          (*PSET :MIN DISTANCE-OF-CONNECTED-CITY
                      NEW-CITY-DISTANCE-FROM-START
                      CONNECTED-CITY-TO)))
    ;;(4) All cities set their distance from the START city to the minimum of
    ;;    their current distance and the value sent by the previous step.

    (*WHEN CITY-P
      (SETQ ANY-NEW-DISTANCE-SHORTER-P (*OR (<!! NEW-CITY-DISTANCE-FROM-START
   CITY-DISTANCE-FROM-START))))
      (*SET CITY-DISTANCE-FROM-START (MIN!! CITY-DISTANCE-FROM-START
                                            NEW-CITY-DISTANCE-FROM-START)))


    ;;(5) If any city got a newer mimimum distance, then go cycle back to
    ;;    step (2) again.

    WHILE ANY-NEW-DISTANCE-SHORTER-P))

  )

;;;
;;; This function just simply pulls the distance of the STOP city from the
;;; CMS.
;;;
(DEFUN PRINT-SHORTEST-PATH (STOP-CITY)
  (FORMAT T "~% The distance from the START city to the stop city is ~d."
        (PREF CITY-DISTANCE-FROM-START
              (GETHASH STOP-CITY PROCESSORS-FOR-CITIES))))
```

Appendix B

List of ESSENTIAL *LISP Commands

This section contains a list of all ESSENTIAL *LISP commands described in this manual, grouped according to functionality. For a more detailed description of any command, refer to the indicated page number.

Configuration Constants:

`*number-of-processors-limit*, page 11`

`*log-number-of-processors-limit*, page 12`

`*number-of-dimensions*, page 12`

`t!!, page 12`

`nil!!, page 12`

Pvar Operations:

`*defvar symbol &optional pvar documentation-string, page 13`

`allocate!! &optional pvar, page 14`

`*deallocate pvar, page 14`

`pvarp object, page 14`

`type-of!! pvar, page 14`

Temporary Allocation:

```
*let ({(symbol &optional pvar)}*) &rest body, page 15

*let* ({(symbol &optional pvar)}*) &rest body, page 15
```

Setting the Contents of a Pvar:

```
*set {pvar-1 pvar-2}*, page 16
```

Reading and Writing the Memory:

```
pref pvar address, page 16
(setf (pref pvar address) lisp-expression)

pref-grid pvar &rest addresses, page 16
(setf (pref-grid pvar &rest addresses) lisp-expression)
```

Selecting the Active Processors:

```
*all &rest body, page 18

*when pvar &rest body, page 18

*cond {(pvar {form}*)}*, page 18

*if pvar then-form &optional else-form, page 19

with-css-saved {form}*, page 19

do-for-selected-processors (symbol) &rest body, page 19
```

Predicate !! Functions:

```
=!!  numeric-pvar &rest numeric-pvars, page 21

/=!!  numeric-pvar &rest numeric-pvars, page 21

<!!  numeric-pvar &rest numeric-pvars, page 22

>!!  numeric-pvar &rest numeric-pvars, page 22
```

<=!! numeric-pvar &rest numeric-pvars, page 22

>=!! numeric-pvar &rest numeric-pvars, page 22

Bit Manipulation !! Functions:

lognot!! integer-pvar, page 22

logior!! &rest integer-pvars, page 22

logxor!! &rest integer-pvars, page 22

logand!! &rest integer-pvars, page 23

logeqv!! &rest integer-pvars, page 23

Boolean !! Functions:

and!! &rest pvars, page 23

or!! &rest pvars, page 23

xor!! &rest pvars, page 23

eql!! pvar1 pvar2, page 23

eq!! pvar1 pvar2, page 23

not!! pvar, page 24

integerp!! pvar, page 23

floatp!! pvar, page 23

numberp!! pvar, page 24

Numerical !! Functions

`!!  lisp-expression, page 24`

`+!!  &rest numeric-pvars, page 24`

`-!!  numeric-pvar &rest numeric-pvars, page 24`

`*!!  &rest numeric-pvars, page 24`

`/!!  numeric-pvar &rest numeric-pvars, page 24`

`1+!!  numeric-pvar, page 24`

`1-!!  numeric-pvar, page 24`

`min!!  numeric-pvar &rest numeric-pvars, 24`

`max!!  numeric-pvar &rest numeric-pvars, 25`

`mod!!  numeric-pvar integer-pvar, 25`

`truncate!!  numeric-pvar, page 25`

`float!!  numeric-pvar, page 25`

`sqrt!!  non-negative-pvar, page 25`

`random!!  limit-pvar, page 25`

Miscellaneous !! Functions

`load-byte!!  from-pvar position-pvar size-pvar, page 25`

`deposit-byte!!  into-pvar position-pvar size-pvar byte-pvar, page 25`

`if!!  pvar then-pvar else-pvar, page 26`

`cond!!  {(pvar {form}*)}*, page 26`

`enumerate!!  pvar, page 26`

*User !! and * Functions:*

***defun, page 27**

***funcall function &rest arguments, page 27**

***apply function arg &optional more-args, page 27**

Parallel Memory Operations:

pref!! pvar-expression cube-address-pvar, page 28
(setf (pref!! dest-pvar-expression cube-address-pvar) value-pvar)

pref-grid!! pvar-expression &rest pvar-addresses &key border-pvar, page 28

pref-grid-relative!! pvar &rest relative-pvar-addresses &key border-pvar, page 29

***pset combinator value-pvar dest-pvar cube-address-pvar, page 29**
where combinator is one of :default, :overwrite, :or, :and, :logior, :logand, :add, :min, :max

***pset-grid combinator value-pvar dest-pvar &rest grid-address-pvars, page 30**

Address Translation and Related Functions:

dimension-size dimension, page 12

self-address!!, page 31

self-address-grid!! dimension-pvar, page 31
grid-from-cube-address cube-address dimension, page 31

cube-from-grid-address address-pvar &rest address-pvars, page 32

grid-from-cube-address!! cube-address-pvar dimension-pvar, page 32

cube-from-grid-address!! address-pvar &rest address-pvars, page 32

Global Operations:

`*logior integer-pvar`, page 33

`*logand integer-pvar`, page 33

`*min numeric-pvar`, page 33

`*max numeric-pvar`, page 33

`*or pvar`, page 33

`*and pvar`, page 33

`*sum numeric-pvar`, page 33

`*product numeric-pvar`, page 34

Initialization:

`*cold-boot &key initial-dimensions`, page 35

`*warm-boot`, page 36

Bibliography

[Hillis 85] W. Daniel Hillis. *The Connection Machine.* MIT Press (Cambridge, Massachusetts, 1985).

[Steele 84] Guy L. Steele, Jr. *Common Lisp: The Language.* Digital Press (Billerica, Massachusetts, 1984).