Handout # 75
15 December 1988

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science

6.821 Programming Languages

*Lisp Reference Manual

Version 5.0
October 1988

Thinking Machines Corporation
Cambridge, Massachusetts

Contents

Preface

Objectives of This Manual

The **Lisp Reference Manual* describes the essential constructs of the *Lisp language and explains the concepts used in programming the Connection Machine in *Lisp.

Intended Audience

The reader is assumed to have a working knowledge of Common Lisp, as described in *Common Lisp: The Language*, and a general understanding of the Connection Machine system. The *Connection Machine Front-End Subsystems* provides the necessary background information on the Connection Machine system.

Revision Information

This manual is a revision of the **Lisp Reference Manual,* Version 4.0, published October 1987. This revision corrects information presented in that version and updates descriptions to account for the implementation of *Lisp Version 5.0. It does *not* fully describe *Lisp Version 5.0; the *Supplement to the *Lisp Reference Manual* provides information on language features new with Version 5.0.

Overview of Manual

Chapter 1. Introduction

Chapter 2. Overview of *Lisp
This chapter provides an overview of the Connection Machine computer and of *Lisp, including:

- A description of the language's basic concepts, such as parallel variables and the selection of particular processors
- The parts of a typical *Lisp program
- Code examples that illustrate using parallel variables and processor selection, defining parallel functions, and performing interprocessor communication

Names that stand for pieces of code (metavariables) appear in italics. In function or macro descriptions, the names of the arguments appear in italics.

Argument names can restrict the type of an argument; argument names that end in the suffix *pvar* must be parallel variables. For example, the name *integer-pvar* restricts the argument to a parallel variable whose fields in the currently selected set of processors must all contain integers.

Braces followed by a star (as in {*symbol*}*) are used as in *Common Lisp: The Language* to indicate the *symbol* may appear zero or more times.

Related Documents

- ******Lisp Release Notes*, Version 5.0
  The current release notes provide a succinct overview of the changes made to to *Lisp since the release of Version 4.3. These are essential reading.

- *Supplement to the *Lisp Reference Manual* , Version 5.0
  This manual updates the **Lisp Reference Manual*, adding descriptions of all features new with the release of *Lisp Version 5.0.

- **Lisp Compiler Guide*, Version 5.0
  This manual describes the current implementation of the *Lisp compiler.

- *Connection Machine Front-End Subsystems*
  The manuals in this volume should be read before the **Lisp Reference Manual*. It explains the configuration of the Connection Machine system, and how to access the Connection Machine from a front-end computer.

- *Connection Machine Parallel Instruction Set*
  The **Lisp Reference Manual* explains how to call Paris from *Lisp. Users who wish to do so should refer also to the Paris manual.

- *Common Lisp: The Language* by Guy L. Steele Jr. Burlington, Mass.: Digital Press, 1984.
  This book defines the de facto industry standard Common Lisp.

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 backtrace, 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 02142-1214
Internet Electronic Mail:	customer-support@think.com
Usenet Electronic Mail:	harvard!think!customer-support
Telephone:	(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:   bug-connection-machine@think.com
```

Please supplement the automatic report with any further pertinent information.

Chapter 1

Introduction

*Lisp (pronounced *star lisp*) is a data parallel language designed to program the Connection Machine.

A Connection Machine (CM) 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 CM configuration can have up to 65,536 processors, each with 4K bits (in model CM-1) or 64K bits of memory (in model CM-2). Typical applications use data types that have components spanning many Connection Machine processors. *Lisp provides the means for creating and manipulating these parallel types.

*Lisp is an extension of Common Lisp. *Lisp adds a new data structure and extensions of many Common Lisp functions that execute in many Connection Machine processors in parallel.

*Lisp has several important features:

- Many *Lisp language features map directly onto Connection Machine instructions; therefore, users quickly develop an intuition for predicting program performance.
- *Lisp includes an interface that permits direct calls to Paris from within a *Lisp program.
- A *Lisp compiler is provided (described in the **Lisp Compiler Guide*).
- A *Lisp simulator is provided for preliminary program testing and debugging. Executing entirely on a serial front end, it simulates the Connection Machine operations.

Chapter 2
Overview of *Lisp

This chapter introduces the main concepts and terminology of *Lisp. It then provides a brief overview, with code examples, of *Lisp operations. All operations appearing in this chapter are described more fully in subsequent chapters.

The primary concepts of the *Lisp language follow:

- *Lisp programs execute on a front-end computer, typically a Symbolics Lisp machine or a Digital Equipment Corporation VAX. As a side effect of running the *Lisp program, the front-end computer generates Paris instructions for the Connection Machine processors to execute. Every so often, the front-end computer transfers data or results of computations to or from the Connection Machine.

- *Lisp programs refer to and manage memory in the Connection Machine processors through Lisp objects called *pvars* (pronounced *pee-vars*, for *parallel variables*). These objects contain information about memory locations in the Connection Machine processors and the possible types of values stored at those locations. A pvar looks like a large vector of Common Lisp values, with each value stored in a different Connection Machine processor. These values may be integers, floating-point numbers, booleans, or any other Lisp object (for example: `0`, `102`, `-5`, `10.333`, `t`, `nil`, `hi-there`). As with Common Lisp, the *Lisp programmer need not be concerned with type coercion, since it is done automatically.

- *Lisp programs control the set of Connection Machine processors that are executing instructions. This set may range from all processors in the machine to none of the processors.

Given these concepts, a *Lisp program typically consists of these parts:

- Permanent Connection Machine storage declarations.

- Code for creating static data structures in Connection Machine memory. This often involves substantial transfers of data from the front-end computer to the Connection Machine.

Pvar Component A single instance among the collection of values represented by a pvar. A pvar component may be any Lisp value; the set of components represented by a pvar need not be all of the same data type.

Field The memory occupied by all the components of a pvar. A field is a string of contiguous bits in the same memory location in *each* processor. The components of a pvar all occupy the same amount of memory (even if they are of different types), and they are all stored at the same memory address in the respective processors.

Pvar Contents The set of values (components) represented by a pvar. These values are stored in the field in the Connection Machine that is described by the pvar.

Currently Selected Set Most *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 special forms, such as ***all**, ***when**, ***cond**, and ***if**.

!! The names of functions and macros that return pvars as their values end with !!. This suffix, pronounced *bang-bang*, is meant to look like two parallel lines. We recommend that user-defined functions follow this convention (although nothing enforces it), because it helps ensure that pvars are produced only in contexts where they can be used. It is an error to produce pvars in contexts where they cannot be used (see chapter 8).

There are a few *Lisp macros whose names do not end in !!, such as ***when**, ***all** and ***let**, that, nonetheless, may optionally return a pvar.

***** All *Lisp functions that perform parallel computation and do not end in !! begin with * (pronounced *star*); hence, the name *Lisp.

Parallel Equivalent of This phrase is used to describe a *Lisp function with reference to a Common Lisp function. For example, "**mod!!** is the parallel equivalent of the Common Lisp **mod**." This means that **mod!!** performs the same calculation as **mod**, only **mod!!** performs the operation in parallel using each component of an argument pvar.

2.2 *Lisp Concepts

This section contains sample code that illustrates some common *Lisp expressions. All the functions used are described fully in later sections of this manual.

The above sets the contents of pvar **a** to 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 a floating-point value, and the result is a floating-point value as well.

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.

2.2.2 Predefined Pvars

Two pvars are predefined in *Lisp. The pvar **t!!** contains the Lisp symbol **t** in each processor; the symbol **t!!** is equivalent to **(!! t)**. Similarly, the pvar **nil!!** contains the Lisp symbol **nil** in each processor.

2.2.3 Selection

When the Connection Machine is initialized, every processor will be in a state to execute all *Lisp instructions in parallel. However, it may be necessary to execute instructions in some subset of the Connection Machine processors.

One way of temporarily selecting a subset is to wrap the ***when** macro around a body of forms. For example, to select the set of all processors whose cube addresses (contained in the pvar returned by the function **self-address!!**) end in 1, one might use the following:

```
;; Create a pvar that is True in all odd processors
(*defvar odd-address-p
         (=!! (!! 1) (mod!! (self-address!!) (!! 2))))

;; Now select all processors with odd cube addresses

(*when odd-address-p ...)
```

In another case, it may be desirable to perform an operation in the processors in which the cube address is even and 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))))
```

2.2.4 *Defun

To define functions that can take pvars as arguments or return them as values, use ***defun** instead of **defun**. For example, 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 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, the following:

```
(defun *compose (*f *g x)
   (*funcall *f (*funcall *g x)))
(*set a (*compose ´sqrt!! ´1+!! (!! 8)))
```

acts like:

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

2.2.5 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 (or

Chapter 3

The Pvar Data Structure

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

3.1 Creating New Pvars

To create a permanent, named pvar, use ***defvar** (analogous to the Lisp **defvar**). To create a permanent, unnamed pvar, use **allocate!!**.

***defvar** *symbol* **&optional** *initial-value-pvar documentation-string* [*M*acro]

This creates a new pvar that is permanently allocated. *Symbol* contains the allocated pvar. The optional argument *initial-value-pvar* may be any pvar or pvar expression. ***defvar** creates a new pvar, initializes it to the contents of *initial-value-pvar* , and sets the *symbol* to that new pvar using **setq**. If no *initial-value-pvar* argument is given, the *symbol* contains a pvar whose values are uninitialized. Note that ***cold-boot** resets 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)
```

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 *Lisp macros for performing this operation are ***let** and ***let***.

***let** ({(*symbol* **&optional** *pvar-expression*)}*) **&rest** *body* [*Macro*]

The first expression following the ***let** should be a list of lists, each specifying one temporary pvar. The elements of each sublist should consist of a Lisp symbol whose value will be the temporary pvar, followed by an optional pvar or pvar expression that will be copied into the new one.

These pvars survive only for the extent of the form. It is an error to try to refer to these pvars outside 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.

***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** is *not* able to return multiple values.

let ({(*symbol* **&optional** *pvar-expression*)}*) **&rest** *body* [*Macro*]

This macro behaves in the same manner as ***let** except that, as in Common Lisp, the defining expressions are evaluated in sequence, so that previous bindings affect the evaluation of future initialization forms.

Some example expressions are:

```
(*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,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))                       ;This does not return a pvar
```

3.3 Setting the Values of Pvars

The ***set** macro allows the contents of one pvar to be set to the contents of another. The destination field is set in those processors that are currently selected. A ***set** expres-

This function returns, as a Lisp value, the component of the field specified by *pvar* in the processor whose grid address is given by *addresses*. There must be as many *address* values as there are dimensions in the processors' current configuration (as specified previously 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)
```

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

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

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

3.5 Declaring Pvar Types

*Lisp does not require that the programmer declare the type of a pvar's contents, nor does it require that all of a given pvar's values be of the same type. A pvar can have an integer in one processor and a floating-point number in the next. This flexibility comes at the cost of decreased efficiency.

Type declarations are a method to reduce runtime overhead for *Lisp code running either interpreted or compiled. Using pvars of defined types results in faster interpreted code and allows the *Lisp compiler to translate *Lisp code into Lisp/Paris.

For more information on pvar types, see chapter 8 of the *Supplement to the *Lisp Reference Manual*. For more information on how the *Lisp Compiler uses pvar types see the **Lisp Compiler Guide*.

3.5.1 Syntax of Declarations

The pvar types supported by *Lisp are signed and unsigned integers, floating-point numbers of varying precision, complex numbers containing floating point data of varying precision, characters, string–chars, and booleans. Pvar type declarations presently are processed only by ***proclaim**, ***let**, ***let***, ***defun**, ***locally** and **the**. These declarations have the same syntax as declarations in Common Lisp. The type of a pvar is specified in the following manner:

(pvar *pvar–type–specifier*)

```
(*proclaim '(type (pvar boolean) finished-p))
(*defvar finished-p nil!!)


(*let* ((temp1 (load-byte!! foo (!! 0) (!! 9)))
        (temp2 (sqrt!! temp1)))
  (declare (type (pvar (unsigned-byte 9)) temp1)
           (type (pvar single-float) temp2))
  ;; temp1 may contain values between 0 and 511.
  ;; temp2 may contain single-floats.
  ...)


(setq xx (allocate!! (!! 1.23) 'xx '(pvar double-float)))
```

*Lisp allows certain elements of declarations to be computed at run time as opposed to compile time. All the *length* arguments to the type declarations may be either constants or run-time expressions such as global configuration variables. Following is a typical example using a global configuration variable:

```
(*let ((temp (self-address!!)))
  (declare
   (type (pvar (unsigned-byte *current-send-address-length*))
         temp))
  ...
 )
```

Chapter 4

Processor Selection

When the Connection Machine is initialized, every processor is in a state to execute all *Lisp instructions in parallel. However, it may be necessary to execute instructions in some subset of processors. In fact, most operations are executed in a subset of Connection Machine processors, which is known as the *currently selected set*.

Some of the macros in *Lisp that change the currently selected set are ***all**, ***when**, ***cond**, and ***if**. These macros 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. Although these macros may modify the currently selected set, they all obey the discipline of restoring the currently selected set to its previous state upon completion. Also, they may be nested as deeply as desired.

It is sometimes useful 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 perform a given operation. The body of these forms is *always* executed, even if there are no selected processors.

Note: In the current implementation, of the forms below that return values, none are configured to allow the return of multiple values. It is an error to attempt to return multiple values from any of these forms.

***all &body** *body* [*Macro*]

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* **&body** *body* [*Macro*]

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

```
(*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))

(*when a (*set b (-!! b))

(*defun f (x y)
   "Returns y divided by x for y greater than 0.
    Returns nil if any x is 0.  The return value is
    undefined a processors where y<0"
   (block foo
      (with-css-saved
         (*when (>!! y (!! 0))
            (if (*or (=!! (!! 0) x))
               (return-from foo nil)
               (/!! y x))
              ))))
```

Chapter 5

Computations on Pvars

This chapter introduces a variety of functions that work within each processor and that return a pvar containing all the processors' results. Recall that in *Lisp, it is conventional for functions that return a pvar to have the suffix !! on their names.

5.1 Predicate Operations

The *Lisp predicate functions are used in the same way as Common Lisp predicates, for instance, in conditional expressions. They return a pvar that contains a t in all processors of the currently selected set in which the predicate holds, and a nil in those in which it does not.

oddp!! *integer-pvar* [*Function*]

The pvar returned by this predicate contains t for each processor where the value of the argument *integer-pvar* is odd, and nil in all others. It is an error if any component of *integer-pvar* is not an integer.

evenp!! *integer-pvar* [*Function*]

The pvar returned by this predicate contains t for each processor where the value of the argument *integer-pvar* is even, and nil in all others. It is an error if any component of *integer-pvar* is not an integer.

plusp!! *number-pvar* [*Function*]

The pvar returned by this predicate contains t for each processor where the value of the argument *number-pvar* is greater than zero, and nil in all others.

/=!! *numeric-pvar* **&rest** *numeric-pvars* *[Function]*

This returns a pvar that contains **t** in each processor where the argument pvars contain unequal values and **nil** elsewhere. If only one argument is given, the returned pvar is **t!!**.

<!! *numeric-pvar* **&rest** *numeric-pvars* *[Function]*

This returns a pvar that contains **t** in each processor where the argument pvars contain values which are in strictly increasing order and **nil** elsewhere. If only one argument pvar is given, the returned pvar is **t!!**.

>!! *numeric-pvar* **&rest** *numeric-pvars* *[Function]*

This returns a pvar that contains **t** in each processor where the argument pvars contain values which are in strictly decreasing order and **nil** elsewhere. If only one argument pvar is given, the returned pvar is **t!!**.

<=!! *numeric-pvar* **&rest** *numeric-pvars* *[Function]*

This returns a pvar that contains **t** in each processor where the argument pvars contain values which are in non-decreasing order and **nil** elsewhere. If only one argument pvar is given, the returned pvar is **t!!**.

>=!! *numeric-pvar* **&rest** *numeric-pvars* *[Function]*

This returns a pvar that contains **t** in each processor where the argument pvars contain values which are in non-increasing order and **nil** elsewhere. If only one argument pvar is given, the returned pvar is **t!!**.

5.1.1 Predefined Pvars

t!! *[Constant]*

This is a pvar whose contents in each processor is the Lisp symbol **t**.

nil!! *[Constant]*

This is a pvar whose contents in each processor is the Lisp symbol **nil**.

It is an error to use **t!!** or **nil!!** as the destination for ***set**, ***pset** or any other form which modifies its argument.

lognot!! *integer-pvar* *[Function]*

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

logior!! &rest *integer-pvars* *[Function]*

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* *[Function]*

This is the parallel equivalent of the Common Lisp function *logxor*. If there are no *integer-pvars*, then **(!! 0)** is returned.

logand!! &rest *integer-pvars* *[Function]*

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

logeqv!! &rest *integer-pvars* *[Function]*

This is the parallel equivalent of the Common Lisp function *logeqv*. If no *integer-pvars* are given, then **(!! -1)** is returned.

5.4 Numerical Operations

This section describes the elementary numerical functions. As with Common Lisp, the results of these functions are always numerically correct. For example, the result of an addition is never truncated, no matter how much memory is required to represent the result. If not enough memory is available, an error is signaled. However, the numerical accuracy of certain arithmetic operations on floating point data is subject to restrictions noted in the *Version 5.0 *Lisp Release Notes*. A few arithmetic operations are also restricted when operating on integer data to a maximum number of bits for each argument. The Release Notes also describe these limitations.

These functions each return results of the same type as the most expensive of their arguments (e.g. if all arguments are integers, the result is generally an integer; but if any argument is a float, the result is a float).

mod!! *numeric-pvar integer-pvar* *[Function]*

This is the parallel equivalent of the Common Lisp function **mod**. It is an error if *integer-pvar* contains zero in any processor.

ash!! *integer-pvar count-pvar* *[Function]*

This is the parallel equivalent of the Common Lisp function **ash**.

truncate!! *numeric-pvar* **&optional** *divisor-numeric-pvar* *[Function]*

This is the parallel equivalent of the Common Lisp function **truncate**, except that only one value (the first) is computed and returned.

round!! *numeric-pvar* **&optional** *divisor-numeric-pvar* *[Function]*

This is the parallel equivalent of the Common Lisp function **round**, except that only one value (the first) is computed and returned.

ceiling!! *numeric-pvar* **&optional** *divisor-numeric-pvar* *[Function]*

This is the parallel equivalent of the Common Lisp function **ceiling**, except that only one value (the first) is computed and returned.

floor!! *numeric-pvar* **&optional** *divisor-numeric-pvar* *[Function]*

This is the parallel equivalent of the Common Lisp function **floor**, except that only one value (the first) is computed and returned.

sqrt!! *non-negative-or–complex–pvar* *[Function]*

This returns the non-negative square root of its argument, if the argument is not complex. If the argument is complex, the principal square root is returned. Unlike Common Lisp., it is an error to provide a negative, non–complex value to **sqrt!!**.

isqrt!! *non-negative-integer-pvar* *[Function]*

This is the parallel equivalent of the Common Lisp function **isqrt**.

random!! *limit-pvar* *[Function]*

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

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* [*Function*]

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))** returns **(!! 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 *size-pvar* must be pvars containing non-negative integers only.

if!! *pvar then-pvar* **&optional** *else-pvar* [*Macro*]

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**. The *else–pvar* argument defaults to **nil!!**. For the execution of the *then-pvar* expression, the currently selected set is set to all processors that passed the predicate, whereas for the execution of the *else-pvar* the currently selected set is set to all the selected processors that failed the predicate. (See also ***if**, which is executed only for side effect.) This is 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** *name arg–list* **&body** *body* *[Macro]*

This is analogous to the Common Lisp **defun** and can be used in place of it in defining user functions that might take as an argument a pvar or that might return a pvar as a result. Using ***defun** is only required if a function is to take pvar arguments and possibly return a non–pvar result. ***defun** returns, as a symbol, the name of the function being defined. Like the Common Lisp **defun**, the body may contain declarations and a documentation string. In particular type declarations for pvar arguments may be provided within the body of a ***defun**.

If, in a given file, a function **foo** defined by ***defun** is called before it is defined textually in the file, or is called but is not defined in the current file, then the user must declare that **foo** is actually a function defined by ***defun** and is not a regular function defined by **defun**. One makes such a declaration with the *Lisp macro ***proclaim**. For example:

```
(*proclaim '(*defun foo))
```

Failure to make such declarations results in incorrectly compiled code.

***funcall** *function* **&rest** *arguments* *[Macro]*

This is used just like Common Lisp's **funcall**, but with functions defined using ***defun**. **One may not use funcall with a function defined using *defun.**

***apply** *function arg* **&optional** *more-args* *[Macro]*

This is used just like Common Lisp's **apply**, but with functions defined using ***defun**.

5.7 Debugging Tools

pretty-print-pvar *pvar* [Macro]
    &key (**:mode *ppp–default–mode***)
        (**:format *ppp–default–format***)
        (**:per-line *ppp–default–per–line***)
        (**:start *ppp–default–start***)
        (**:end *ppp–default–end***)

This prints out the contents of a pvar in all processors. If **:per-line** is **nil**, no newlines are ever printed between values; otherwise, **:per-line** values are printed out and then a newline is output. The keyword **:mode** can have the value **:cube** or **:grid**; in the latter

list-of-active-processors *[Function]*

This simply returns a list of cube addresses of all the currently selected processors. The order of this list is not specified. Since this function is so useful, an alias, **loap**, is also defined. This could be written as:

```
(defun list-of-active-processors ()
   (let ((return-list nil))
      (do-for-selected-processors (processor)
         (push processor return-list))
      return-list))
```

pretty-print-pvar-in-currently-selected-set *pvar*
  &key *format start end* *[Function]*

This function prints out the the cube address and value of *pvar* for all processors in the currently selected set. Since this function is so useful, an alias, **ppp-css**, is also defined. *format* defaults to **"~s"**. This function returns no values.

6.4 Global Operations

The following functions reduce the contents of a pvar in all selected processors into a single Lisp value, which is then returned:

***logior** *integer-pvar* [**Defun*]

This returns a Lisp value that is the bitwise logical inclusive or 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* [**Defun*]

This returns a Lisp value that is the bitwise logical and 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* [**Defun*]

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* [**Defun*]

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* [**Defun*]

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 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* [**Defun*]

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.

grid-from-cube-address!! *cube-address-pvar dimension-pvar* [*Function*]

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* [*Function*]

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

off-grid-border-p!! **&rest** *grid-address-pvars* [*Function*]

This function returns a boolean pvar that is true if the *grid-address-pvars* specify an address that is invalid given the current dimensions, and false otherwise. It is an error for any component of *grid-address-pvar* to be a non-integer.

off-grid-border-relative-p!! **&rest** *relative-grid-address-pvars* [*Function*]

This function is identical to **off-grid-border-p!!** except that the *relative-grid-address-pvars* specify relative addresses.

Chapter 7
Using the Connection Machine

The *Lisp language resides in a package named *LISP. To use the language, the user must either be in that package:

```
(in-package '*LISP)
```

or else make that package available to the package the user is in:

```
(use-package '*LISP)
```

On Symbolics Lisp machines, the user should put the following package attribute in the attribute list of any file that contains functions to be put in the ***LISP** package:

```
Package: (*LISP COMMON-LISP-GLOBAL)
```

For instructions on how to load the *Lisp language into your Lisp machine, please refer to the *Connection Machine Front-End Subsystems*.

7.1 Using the Connection Machine Hardware

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

Note: The macro ***cold–boot** has been enhanced to work with n–dimensional NEWS, *Lisp interpreter safety, and geometry objects. These changes and the new features to which they are related are all documented in the *Supplement to the *Lisp Reference Manual.*

***cold-boot &key :initial-dimensions** *initial-dimensions* [*Macro*]

This function initializes *Lisp and must be called immediately after loading in the *Lisp software. It resets the internal state of the *Lisp system and of the Connection

7.1.1 Initialization Lists for *cold-boot and *warm-boot

Users can define a set of forms to be executed automatically before and after each execution of *cold-boot and *warm-boot. These user-defined initialization lists are stored in one or more of these variables:

before-*cold-boot-initializations [*Variable*]

after-*cold-boot-initializations [*Variable*]

before-*warm-boot-initializations [*Variable*]

after-*warm-boot-initializations [*Variable*]

New forms are added using the function **add-initialization**, and removed using **delete-initialization.**

add-initialization *name-of-form form variable* [*Function*]

The argument *name-of-form* is a character string that names the form being added. The argument *form* may be any executable Lisp form. Adding two forms with the same name is permissible only if the forms are the same according to the function **equal**; otherwise an error is signaled. The *variable* should be one of the initialization-list variables above, or it may be a list of such variables, in which case the *form* is added to each initialization list named. The *form* and *variable* arguments must be quoted so that they are not evaluated during the call to **add-initialization**. For example:

```
(add-initialization (string 'items)
                    '(initialization-items)
                    '*after-*warm-boot-initializations*)
```

delete-initialization *name-of-form variable* [*Function*]

This function deletes the form named by *name-of-form* from the initialization list (or lists) specified by *variable*. The arguments are specified in the same manner as the first and third arguments for **add-initialization**. For example:

```
(delete-initialization (string 'items)
                       '*after-*warm-boot-initializations*)
```

boot and ***warm-boot** work in the same manner as for the interpreter. The simulator is more lenient than the interpreter with respect to dimensioning the machine: the simulator allows arbitrary extent in each dimension, while the interpreter currently supports only extents that are powers of two. Warnings are issued if one tries to configure the simulator in a way that the interpreter cannot handle.

Since the simulator and interpreter implementations of *Lisp are very different, it is unfortunately necessary to recompile code when switching from one system to the other. It is not possible to load the simulator once the interpreter version of *Lisp has been loaded, and vice versa.

The symbols ***lisp-hardware** and ***lisp-simulator** are appended to the Common Lisp ***features*** list when running on the interpreter and on the simulator, respectively. These symbols can be used to perform read-time conditionalization of code.

7.2 Interfacing Paris Code to *Lisp

It is sometimes necessary to explicitly call Paris instructions from within *Lisp programs.

Paris instructions require pvars' memory addresses and lengths as their arguments. While a pvar is commonly—and appropriately—regarded as a "parallel variable," that is, a program variable that has a value in each CM processor, a pvar actually exists in the front end as a Lisp object that points to and describes a field in each CM processor's memory. These fields in the CM contain the values that comprise the parallel variable. The front-end Lisp object also contains pvars' addresses and lengths, information that may be needed as arguments to Lisp/Paris functions. The following functions return this information.

```
(pvar-location pvar)
(pvar-length pvar)
```

These functions return the location (address) and length of a pvar.

```
(pvar-type pvar)
```

This function returns the type of a pvar. The type of a pvar may be one of **:general**, **:field**, **:signed**, **:float**, **:boolean**, **:complex**, **:character**, **:string-char**, **:array**, or **:structure**. These correspond to pvar types **t** (for general), **unsigned-byte**, **signed-byte**, **defined-float**, **boolean**, **complex**, **character**, **string-char**, **array**, and types defined by ***defstruct**, respectively.

Chapter 8

Avoiding Potential Difficulties

This chapter describes potential difficulties in using *Lisp and ways the user can avoid them.

8.1 Pvar Values in Non-Selected Processors

It is an error to depend on the value of a pvar in a processor that was not in the currently selected set at the time the pvar was created. For instance, the following is incorrect:

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

The *Lisp language definition does not define the value printed in the above example because the pvar **foo** was only given values in the active processors, 0 through 9.

8.2 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)))
```

8.5 *cold-boot

***cold-boot** calls **cm:coldboot** to initialize Connection Machine state and also initializes the *Lisp run–time systems. This function *must* be called when first entering the *Lisp environment (hardware or simulator) if the user intends to evaluate any code. In addition, we strongly recommend that ***cold-boot** be called between runs of application programs. Finally, ***cold-boot** should be called if ***warm-boot** has failed to clear an error state.

***cold-boot** is intended to be used as a top-level form. Under no circumstances should it be used inside of a ***defun** form, either lexically or in such a manner that it might be executed while a ***defun** function is being evaluated.

8.6 pref!!, pref-grid!! and pref-grid-relative!!

These *Lisp functions (which are actually macros) are defined to evaluate their source argument(s) in the context of the set of addresses defined by evaluating their address pvar(s). Therefore, writing a function **foo** that calls one of these functions with a source argument *s* passed into **foo** may not work because *s* will have already been evaluated by the Lisp evaluator before the body of **foo** is evaluated.

The solution is to define such functions as macros. For example:

```
(*defun foo (condition source-pvar address-pvar)
   (if condition
       (pref!! source-pvar address-pvar)
       (!! 0)
   ))
```

The above may not work as intended and should be rendered as

```
(defmacro foo (condition source-pvar address-pvar)
   `(if ,condition
        (pref!! ,source-pvar ,address-pvar)
        (!! 0)
    ))
```

If **source-pvar** is always a symbol and not a pvar expression, then this modification is not necessary.

8.7 Multiple Values

The current implementation does not support the return of multiple values from any *Lisp form. It is error to attempt such an operation.

Appendixes

Appendix A

*Lisp Symbols

This appendix lists all *Lisp symbols and pvar types described in this manual.

Index

Index

References are to page numbers. Separate listings of *Lisp symbols and pvar types are provided in appendix A.