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

Preface ................................................................. v
Customer Support ............................................. ix

Chapter 1 Introduction ........................................ 1

Chapter 2 Overview of *Lisp ............................... 3
  2.1 *Lisp Terminology ........................................ 4
  2.2 *Lisp Concepts .......................................... 5
    2.2.1 Pvars .............................................. 6
    2.2.2 Predefined Pvars ............................... 7
    2.2.3 Selection ..................................... 7
    2.2.4 *Defun ........................................ 9
    2.2.5 Communication ............................. 9
  2.3 Configuration Constants ............................ 10

Chapter 3 The Pvar Data Structure .................. 11
  3.1 Creating New Pvars ................................... 11
  3.2 Allocating Local Pvars ............................ 12
  3.3 Setting the Values of Pvars ..................... 13
  3.4 Reading and Writing Fields in Specific Processors 14
  3.5 Declaring Pvar Types ............................ 15
    3.5.1 Syntax of Declarations .................. 15
    3.5.2 Example Declarations .................. 16

Chapter 4 Processor Selection .......................... 19
  5.2 Logical Operations ................................ 26

Chapter 5 Computations on Pvars ................... 23
  5.1 Predicate Operations ................................ 23
    5.1.1 Predefined Pvars ......................... 25
5.2 Logical Operations ................................................................. 26
5.3 Logical Operations on Integers in Pvars ........................... 26
5.4 Numerical Operations ............................................................ 27
5.5 Miscellaneous Operations ..................................................... 30
5.6 User-Defined Operations ....................................................... 32
5.7 Debugging Tools ................................................................. 33

Chapter 6 Communication .................................................. 37
6.1 Communication between Processors ................................. 37
6.2 Block Data Transfer between the Front End 
and the Connection Machine System ................................. 43
6.3 Scan Functions ................................................................. 45
6.4 Global Operations ............................................................. 49
6.5 Processor Addressing .......................................................... 50

Chapter 7 Using the Connection Machine .......................... 53
7.1 Using the Connection Machine Hardware ....................... 53
  7.1.1 Initialization Lists for *cold-boot and *warm-boot ...... 55
  7.1.2 Configuration Variables ........................................... 56
  7.1.3 The *Lisp Simulator .................................................. 56
7.2 Interfacing Paris Code to *Lisp ........................................ 57
7.3 *Lisp Memory Management: Stack and Heap Storage .... 58

Chapter 9 Avoiding Potential Difficulties ............................ 59
9.1 Pvar Values in Non-Selected Processors ......................... 59
9.2 The Extent of Pvars .......................................................... 59
9.3 Using Mapcar on Functions that Return pvars ................ 60
9.4 *warm-boot ................................................................. 60
9.5 *cold-boot ................................................................. 61
9.6 prefix, prefix-grid! and prefix-grid-relative! ..................... 61
9.7 Multiple Values ........................................................... 61
9.8 Grid and Cube Addresses ............................................... 62

Appendix A. *Lisp Symbols .................................................. 65
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
Chapter 3. The Pvar Data Structure

Pointers to Connection Machine memory are stored in Lisp objects called parallel variables, or pvars (pronounced pee-vars). This chapter describes the operations (functions, macros) that create, destroy, assign the contents of, and declare the types of pvars.

Chapter 4. Processor Selection

This chapter describes the operations that select the set of processors to perform a given operation. This set can change from one line of code to another, as determined by the programmer.

Chapter 5. Computations on Pvars

This chapter describes the operations for combining and comparing parallel variables numerically and logically, the operations for defining new functions that return parallel variables or perform parallel computations, and the functions that assist debugging.

Chapter 6. Communication

This chapter discusses the mechanisms for moving data between Connection Machine processors in parallel; block data transfer between the front end and the Connection Machine; scanning functions, in which each processor receives a result based on the values in preceding processors; global operations, which combine data from all Connection Machine processors; and processor address generation and translation.

Chapter 7. Using the Connection Machine

This chapter describes the operations for configuring the Connection Machine system; the configuration constants, which contain information about the Connection Machine's current configuration; and the *Lisp simulator, which allows *Lisp programs to be run and tested on the front-end computer alone. How to call Lisp/Paris (the Connection Machine's assembly language) from within *Lisp programs and *Lisp memory management is also explained.

Chapter 8. Avoiding Potential Difficulties

This chapter points out some potential difficulties and common user errors.

Notation Conventions

The notation conventions in this manual strive to be compatible with those used in Common Lisp: The Language.

Symbol names in running text appear in a bold typeface. For example: *cold-boot. Code examples are printed in a typewriter style typeface:

(cons a b)
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**

  The current release notes provide a succinct overview of the changes made 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.

  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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Information</th>
</tr>
</thead>
</table>
| **U.S. Mail**   | Thinking Machines Corporation  
Customer Support  
245 First Street  
Cambridge, Massachusetts 02142-1214 |
| **Internet**    | customer-support@think.com                                                  |
| **Usenet**      | 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.
- Main body of *Lisp program, which typically contains:
  - Temporary Connection Machine storage allocation
  - Selection of Connection Machine processors for expression evaluation
  - Parallel expression evaluation with the result stored in a destination
  - Massive communication of data within the Connection Machine system
  - Transfer of results back to the front-end computer.

2.1 *Lisp Terminology

The following terms are used with specific meanings in *Lisp:

**Processors**  The conceptual entities that operate on data in parallel are called *processors*. Often these correspond to the physical processors, but sometimes a single physical processor simulates the action of several conceptual processors. In this case, the simulated processors are referred to as *virtual processors*. This simulation is transparent to the programmer. In the *Connection Machine Parallel Instruction Set: Paris Reference Manual* is a chapter on concepts, which describes in detail the implications and mechanisms of virtual processors.

**Cube Address** Each processor has a unique *cube address*. Cube addresses range between zero and the number of processors less 1. On a 65,536 processor Connection Machine, the range is 0 to 65,535, although it may be much larger if virtual processors are being used.

**Grid Address** A 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 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.

**Pvar** A Lisp object that represents a collection of values stored one per processor in the Connection Machine. A pvar also holds the information needed by the front end to manage the collection of values.
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.
2.2.1 Pvars

The following code creates 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 pvars have been initialized with specific values: 
- b is a Lisp symbol that contains 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;
- e contains an integer that is the cube address of the next higher processor.

The function pref can be used to read out some of the above values. The arguments to 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, which is the component of pvar c in processor 0.

Similarly,

(pref d 365)

returns the Lisp value t, the component of pvar d in processor 365.

*Lisp uses the macro *setf (akin to the Common Lisp macro setf) to turn accessor expressions into modifier expressions. For example, there is no function that is 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) now returns 15 because 15 is now contained in pvar b in processor 0.

The following examples demonstrate arithmetic operations on the example pvars.

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

\[
(*\text{set} \ a \ (-!! \ b \ (*!! \ a \ (!! \ 2)))
\]

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:

```lisp
;; 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:

```lisp
(*when (and!! (not!! odd-address-p)
 (=!! a (!! 0)))
 (*set a (+!! a b)))
```
or equivalently, (2) nest *when expressions:

\[
(*\text{when} \ (\not!! \ \text{odd-address-p})
\begin{align*}
  (*\text{when} \ (=!! \ a \ (!! \ 0))
    (*\text{set} \ a \ (+!! \ a \ b)))
\end{align*}
\]

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

\[
(*\text{let} \ ((g \ (\text{mod}!! \ (\text{self-address}!!) \ (!! \ 4))))
\begin{align*}
  (*\text{when} \ (=!! \ g \ (!! \ 0))
    (*\text{set} \ a \ (+!! \ a \ b)))
\end{align*}
\]

This code 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:

\[
(*\text{let} \ ((g \ (\text{mod}!! \ (\text{self-address}!!) \ (!! \ 4))))
\begin{align*}
  (*\text{cond}
    \begin{align*}
      & (=!! \ g \ (!! \ 0)) \ (*\text{set} \ a \ (+!! \ a \ b))) \\
      & (=!! \ g \ (!! \ 1)) \ (*\text{set} \ a \ (-!! \ a \ b))) \\
      & (=!! \ g \ (!! \ 2)) \ (*\text{set} \ a \ (*!! \ a \ b))) \\
      & (=!! \ g \ (!! \ 3)) \ (*\text{set} \ a \ (\text{div}!! \ a \ b)))
    \end{align*}
  \end{align*}
\]

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

\[
(*\text{if} \ (<! \ z \ x)
\begin{align*}
  (*\text{set} \ y \ (!! \ 5))
  (*\text{set} \ y \ (!! \ 6)))
\end{align*}
\]

and

\[
(*\text{set} \ y
\begin{align*}
  (if!! \ (<! \ z \ x)
    (!! \ 5)
    (!! \ 6)))
\end{align*}
\]

Note that cond!! and if!! return a pvar that must be stored into some destination, whereas *cond and *if are executed only for side effect.
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:

```lisp
(*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:

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

To pass a *Lisp function as an argument, use *funcall. For example, the following:

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

acts like:

```lisp
(*set a (*compose '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
NEWS for short). 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:

\[
\begin{align*}
(* \text{set} \ a \ (+!! \ (\text{news!!} \ a \ 0 \ 1)) & \quad ; \text{north} \\
\quad (\text{news!!} \ a \ 1 \ 0) & \quad ; \text{east} \\
\quad (\text{news!!} \ a \ -1 \ 0) & \quad ; \text{west} \\
\quad (\text{news!!} \ a \ 0 \ -1))) & \quad ; \text{south}
\end{align*}
\]

Note: The macro \text{news!!} is defined in the *Lisp Reference Manual. It is new with *Lisp Version 5.0.

The following causes the first 100 processors in the Connection Machine to write the contents of field \( b \) into the field \( a \) of those processors whose addresses are 7 larger:

\[
(* \text{when} \ (<!!(\text{self-address!!}) \ (!! 100)) \ ; \text{select first 100 processors}
\quad (* \text{setf} \ (\text{pref!!} \ a \ (+!! \ (\text{self-address!!}) \ (!! 7))) \ b))
\]

Note that \text{pref!!} is the parallel version of \text{pref}; all selected processors perform in parallel a \text{pref} from the processor of their choice.

Finally, to find the single maximum value of \( a \) in all even processors, one possibility is as follows:

\[
(* \text{when} \ (=!! \ (!! 0) \ (\text{mod!!} \ (\text{self-address!!}) \ (!! 2)))
\quad (* \text{max} \ a))
\]

### 2.3 Configuration Constants

*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 runs on machines with differing amounts of physical hardware.

The configuration variables defined in chapter 7 specify the parameters of the machine as perceived by the user's program. If a program uses only these constants and functions, it will run in any configuration.

The size of a simulated configuration of processors is specified through the \text{cold-boot} macro (see chapter 7), which resets the Connection Machine to a known state.
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 [Macro]

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)
*deallocate-*defvars &rest pvar-names

This function permanently deletes the pvars specified in pvar-names. If pvar-names is nil or :prompt, the user is prompted for each pvar ever declared with *defvar. If pvar-names is :all, then all pvars declared with *defvar are deleted after the user is prompted for confirmation. If pvar-names is :all-noconfirm, then all pvars declared with *defvar are deleted. Before using :all, users should be certain that no library functions they call depend on any pvars created with *defvar. (The two predefined pvars, t!! and nil!!, are never deleted.)

Here are some sample uses:

(*deallocate-*defvars 'foo) ;delete foo pvar
(*deallocate-*defvars 'foo 'bar) ;delete foo and bar pvars
(*deallocate-*defvars) ;prompt user for pvars to delete
(*deallocate-*defvars :prompt) ;prompt user for pvars to delete
(*deallocate-*defvars :all) ;delete all pvars declared with *defvar

allocate!! &optional pvar-initial-value name type

This creates a permanent pvar named name and of type type. It is like *defvar, except that the created pvar is simply returned. It is up to the user to store it some place. If no pvar-initial-value is specified, then the returned pvar will have values that are undefined; otherwise, pvar-initial-value is used to initialize the newly allocated pvar.

Examples:

(setq a (allocate!! (!! 5)))
(setq b (allocate!! (!! 5) 'new-pvar 'boolean-pvar))

*deallocate pvar

This deallocates the given pvar if it was permanently allocated (i.e., it was defined using allocate!!). It is an error to use a pvar after it has been deallocated. The order in which pvars are deallocated does not matter. This function returns nil.

pvarp object

This returns t if the argument is a pvar and nil if it is not.

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

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

The first expression following the *let should be a list of lists, each specifying one tem­
porary 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 multi­
ple values.

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

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:

\[
(*\text{let*} (a
  (b (!! 8))
  (c (*!! b (!! 528)))
  (d (!! -2.715))
  (e (self-address!!)))
(some-pvar-function a b c d e) ; This may modify a,b,c,d,or e
(+!! a b c d e)) ; This returns a pvar
\]

\[
(*\text{let} ((a (load-byte!! (self-address!!) (!! 16) (!! 16))))
(*\text{max} a)) ; This does not return a pvar
\]

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
*set takes multiple pairs, which are set sequentially. *set is sometimes used in conjunction with *all to set the contents of one pvar to the contents of another in all processors, not just the selected ones.

\[ \text{*set \{destination-pvar-1 source-pvar-1\}} \]

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

Some examples of the use of *set are:

\[
\begin{align*}
\text{(*set a (+!! b c))} \\
\text{(*all (*set a (!! -1) b a c (!! -3)))} \\
\text{(*when (>!! d (!! 4)) (*set a b))}
\end{align*}
\]

### 3.4 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.

\[ \text{pref pvar address} \]

This macro returns, as a Lisp value, the component 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.

\[
\text{(pref foo 17)}
\]

returns the contents of pvar foo from processor 17.

\[
\text{(*setf (pref foo 17) (* 19 89))}
\]

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

\[ \text{pref-grid pvar &rest addresses} \]

Note: The function pref-grid is obsolete with Version 5.0. See section 8.7 of the Supplement to the *Lisp Reference Manual for information on new, alternative constructs.
This function returns, as a Lisp value, the component of the field specified by \texttt{pvar} in the processor whose grid address is given by \texttt{addresses}. There must be as many \texttt{address} values as there are dimensions in the processors’ current configuration (as specified previously with \texttt{*cold-boot}). \texttt{*setf} may be used with \texttt{pref-grid} to write a value into a single processor of a pvar.

\begin{verbatim}
(pref-grid bar 4 7)
\end{verbatim}

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

\begin{verbatim}
(*setf (pref-grid bar 4 7 8) (* 19 89))
\end{verbatim}

The above sets the component of pvar \texttt{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 \texttt{*proclaim}, \texttt{*let}, \texttt{*let*}, \texttt{*defun}, \texttt{*locally} and \texttt{the}. These declarations have the same syntax as declarations in Common Lisp. The type of a pvar is specified in the following manner:

\begin{verbatim}
(pvar pvar-type-specifier)
\end{verbatim}
Where \textit{pvar-type-specifier} is either a symbol or a list.

\begin{verbatim}
(pvar boolean)
(pvar character)
(pvar string-char)
\end{verbatim}

These forms are used in declarations of pvars containing boolean, character, and string-char elements, respectively.

\begin{verbatim}
(pvar (unsigned-byte length))
\end{verbatim}

Either of these forms declares a pvar to contain a positive integer of \textit{length} bits. For example, if \textit{length} is 8, the pvar may contain integers between 0 and 255. The minimum allowed length is 1 bit.

\begin{verbatim}
(pvar (signed-byte length))
\end{verbatim}

Either of these forms declares a pvar to contain a signed integer of \textit{length} bits. For example, if \textit{length} is 8, the pvar may contain integers between -128 and 127. The minimum allowed length is 2 bits.

\begin{verbatim}
(pvar short-float)
(pvar single-float)
(pvar double-float)
(pvar long-float)
\end{verbatim}

These types describe floating-point numbers of different significand and exponent sizes. The \textit{short-float}, \textit{single-float}, \textit{double-float}, and \textit{long-float} types are standard IEEE formats.

\begin{verbatim}
(pvar (defined-float significand-length exponent-length))
(float-pvar significand-length exponent-length)
\end{verbatim}

It is possible to create a floating-point number with an arbitrary number of significand and exponent bits through the use of \textit{defined-float} or \textit{float-pvar}. See \textit{Connection Machine Parallel Instruction Set} for a discussion on minimum and maximum floating-point number sizes and performance considerations.

For complete information of types and type specifiers, see chapter 8 of the \textit{Supplement to the *Lisp Reference Manual}.

\section*{3.5.2 Example Declarations}

The following examples illustrate the use of the Common Lisp \texttt{type} and \texttt{declare} statements with the above pvar type specifiers:
(*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 con­
stants 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
processors in which \texttt{pvar} is non-nil. \textbf{Even if there are no selected processors, ALL forms in the body are evaluated.} The value of the final expression in the body is returned whether it is a Lisp value or a pvar.

\begin{verbatim}
*if pvar then-form \&optional else-form [Macro]
\end{verbatim}

This form is analogous to the Lisp if. \texttt{then-form} is performed in all processors of the currently selected set in which \texttt{pvar} is not \texttt{nil}. The optional \texttt{else-form} is evaluated in all other processors of the currently selected set in which the \texttt{pvar} is \texttt{nil}. \textbf{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.

\begin{verbatim}
*cond \{(pvar \{form\}*)\} [Macro]
\end{verbatim}

This form is analogous to the Lisp cond. Unlike the Lisp cond, \texttt{*cond} evaluates all clauses; however, the currently selected set is determined by the \texttt{pvar} expressions. The \texttt{n}th consequent is evaluated with a selected set made up of initially selected processors that didn't pass the first \texttt{n-1} tests, but did pass the \texttt{n}th one. \texttt{t!!} selects all remaining processors in the initial selected set. \textbf{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 cond!! in chapter 5).

\begin{verbatim}
with-css-saved \&body body [Macro]
\end{verbatim}

This form is used whenever control flow would abnormally pass out of a \texttt{*Lisp form} that restricts the currently selected set (as when using \texttt{throw}, \texttt{return-from}, or \texttt{go} to leave the body of a \texttt{*when}). \texttt{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 \texttt{with-css-saved} form was begun. \texttt{with-css-saved} returns what is returned by the evaluation of the last form of its body.

\begin{verbatim}
do-for-selected-processors (symbol) \&body body [Macro]
\end{verbatim}

This form evaluates \texttt{body} as many times as there are active processors, each time with \texttt{symbol} bound to the cube address of a different active processor. As with the Common Lisp \texttt{dotimes}, the \texttt{return} function may be used to exit the \texttt{do-for-selected-processors} form immediately. Normally, \texttt{do-for-selected-processors} returns \texttt{nil}.

Some examples of the use of these macros are:

\begin{verbatim}
(*all (*set a b))
\end{verbatim}
Chapter 4. Processor Selection

(*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.

\[
\text{oddp!! integer-pvar} \quad \text{[Function]}
\]

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

\[
\text{evenp!! integer-pvar} \quad \text{[Function]}
\]

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

\[
\text{plusp!! number-pvar} \quad \text{[Function]}
\]

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

The pvar returned by this predicate contains t for each processor where the value of the argument number-pvar is less than zero, and nil in all others. Note: (minusp!! (!! -0.0)) is always false.

eq!! 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.

zerop!! numeric-pvar

This is the parallel equivalent of the Common Lisp function zerop.

=!! numeric-pvar &rest numeric-pvars

This returns a pvar that contains t in each processor where the argument pvars contain numerically equal values 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 is t!!.
Chapter 5. The Pvar Data Structure

/$=!!\quad numeric-pvar \&rest numeric-pvars \quad [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!!.

<!!\quad numeric-pvar \&rest numeric-pvars \quad [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!!.

>!!\quad numeric-pvar \&rest numeric-pvars \quad [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!!.

<=$!!\quad numeric-pvar \&rest numeric-pvars \quad [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!!.

>=$!!\quad numeric-pvar \&rest numeric-pvars \quad [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

$!!!\quad [Constant]
This is a pvar whose contents in each processor is the Lisp symbol t.

nil!!!\quad [Constant]
This is a pvar whose contents in each processor is the Lisp symbol nil.

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

*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.

\[
\text{not!! } p\text{var}
\]

This returns t for all processors in which \textit{pvar} is nil, and nil otherwise.

\[
\text{and!! } \&\text{rest } p\text{vars}
\]

This evaluates the \textit{pvars} from left to right in all selected processors. As soon as one of the \textit{pvars} evaluates to nil in a processor, that processor is removed from the currently selected set for the remainder of the \textit{and!!}. \textit{and!!} returns the value of the last \textit{pvar} for all selected processors in which all the \textit{pvars} are true; it returns nil otherwise. If no \textit{pvars} are given, then t!! is returned.

\[
\text{or!! } \&\text{rest } p\text{vars}
\]

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

\[
\text{xor!! } \&\text{rest } p\text{vars}
\]

This performs the xor function on all the \textit{pvars}. If no \textit{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.

5.3 Logical Operations on Integers in Pvars

This section contains a variety of Boolean functions that operate bitwise on the bits of the fields described by the argument \textit{pvars} and return a \textit{pvar} that holds the result. These functions may be used only on \textit{pvars} whose contents are integers.
Chapter 5. The Pvar Data Structure

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).
!! 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. If there are no arguments, then (!! 0) is returned.

-!! numeric-pvar &rest numeric-pvars

This subtracts the contents of the second through last argument pvars from the contents of the first. If there is only one argument, the result is its negation.

*!! &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 the first pvar by the rest of the pvars. If there is only one argument, the result is the inverse of pvar. Note: /!! always returns a pvar whose contents are all floating-point or complex 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. A new pvar is returned.

1-!! numeric-pvar

This decrements the argument pvar by 1. A new pvar is returned.

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**  
*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.
**signum!! number-pvar**  
[Function]  
This function returns a pvar containing -1, 0, or 1 according to whether the number is negative, zero, or positive. For a floating-point number, the result is a floating-point number of the same format.

**sin!! radians**  
[Function]  
**cos!! radians**  
[Function]  
**tan!! radians**  
[Function]  
The function sin!! returns the sine of the argument, cos!! returns the cosine, and tan!! returns the tangent.

**log!! number &optional base**  
[Function]  
This function returns the logarithm of the argument number in the base base. If base is absent, the natural logarithm is returned.

**float!! number-pvar &optional other-pvar**  
[Function]  
This function converts any number to a floating-point number. In processors in which number-pvar already contains floating-point numbers, those numbers are returned; otherwise, single-float numbers are produced. When the optional argument other-pvar is given, which must contain floating-point numbers, number-pvar is converted to the same format as other-pvar.

**rot!! integer-pvar n-pvar word-size-pvar**  
[Function]  
This function returns integer-pvar rotated left n-pvar bits, or rotated right |n-pvar| bits if n-pvar is negative. The rotation considers integer-pvar as a number of length word-size-pvar bits. This function is especially fast when n-pvar and word-size-pvar are both constant pvars.

**Note:** Many more numeric functions have been added to *Lisp with Version 5.0. See chapters 1 and 7 of the Supplement to the *Lisp Reference Manual.

### 5.5 Miscellaneous Operations

**load-byte!! from-pvar position-pvar size-pvar**  
[Function]  
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 \texttt{position-pvar} and \texttt{size-pvar} are constants, as in \texttt{('?\ lisp-value)}. \texttt{from-pvar} must be a pvar containing integers, while \texttt{position-pvar} and \texttt{size-pvar} must be pvars containing non-negative integers. Out-of-range bits are treated as zero for positive integers (for example, \texttt{(load-byte!! '\(1\) \(2\) \(3\))} returns a pvar that contains zero in each processor), and one for negative integers (for example, \texttt{(load-byte!! '\(-1\) \(2\) \(3\))} returns a pvar that contains 7 in each processor).

\texttt{deposit-byte!! into-pvar position-pvar size-pvar byte-pvar} \hspace{1cm} \textbf{[Function]}

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

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

\texttt{if!! pvar then-pvar \&optional else-pvar} \hspace{1cm} \textbf{[Macro]}

This returns a pvar that contains the contents of the \texttt{then-pvar} in all processors in which \texttt{pvar} is non-nil, and the contents of \texttt{else-pvar} in all processors in which \texttt{pvar} is nil. The \texttt{else-pvar} argument defaults to nil!!!. For the execution of the \texttt{then-pvar} expression, the currently selected set is set to all processors that passed the predicate, whereas for the execution of the \texttt{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:

\begin{verbatim}
(*let ((result)
       (temp-pred pvar))
  (*when temp-pred
    (*set result then-pvar))
  (*when (not!! temp-pred)
    (*set result else-pvar))
  result)
\end{verbatim}

An example that demonstrates the usefulness of \texttt{if!!} is the following function to take the absolute value:
(*defun abs!! (pvar)
  (if!! (>!! pvar (!! 0)) pvar (-!! pvar)))

cond!! (pvar (form)*) [Macro]

If there are no clauses, cond!! returns nil!!. Otherwise, cond!! is roughly equivalent to the following pseudo-code:

(if!! pvar-1
  (progn all-the-forms-for-clausel)
  (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. (See also *cond, which is executed only for side effect).

enumerate!! [Function]

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. The numbers are ordered, 0 being put in the processor with the smallest cube address, 1 being put in the processor with the next smallest cube address, etc. If all processors in the Connection Machine are selected, this is equivalent to the function self-address!!.

rank!! numeric-pvar predicate [Function]

Rank!! returns a pvar containing the values 0 through 1 less than the active number of processors, such that for all values v1 and v2, and for all active processors p1 and p2, if v1 < v2, then the value of numeric-pvar in processor p1 satisfies the serial analog of predicate with respect to the value of numeric-pvar in processor p2. (Currently, predicate must be the symbol <=!!, and its serial analog is the predicate <=.) If there are no active processors, rank!! returns numeric-pvar.

5.6 User-Defined Operations

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 name arg-list &body body

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

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

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

5.7 Debugging Tools

pretty-print-pvar pvar &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
case the pvar is printed out using grid addressing rather than cube addressing. If :start
and/or :end are given, these restrict the range of processors over which values are
printed out. The keyword :format has as its value a string which controls the printing
format for each value; its value is used directly by the Common Lisp format function.

If pretty-print-pvar accesses a processor that has no defined value for pvar, then the
symbol ERROR is printed out.

ppp  
[Macro]
This macro is identical to pretty-print-pvar.

Note: Version 5.0 updates the *Lisp debugging tools. ppp has been augmented and
several related operations have been added. (See chapter 7 of the Supplement to the
*Lisp Reference Manual.)

*ppp-default-mode*  
[Variable]
This variable provides the default value for the keyword argument :mode. Its initial
value is the keyword :cube. Its other legal value is :grid.

*ppp-default-format*  
[Variable]
This variable provides the default value for the keyword argument :format. Its initial
value is the string "-s ".

*ppp-default-per-line*  
[Variable]
This variable provides the default value for the keyword argument :per-line. Its initial
value is nil.

*ppp-default-start*  
[Variable]
This variable provides the default value for the keyword argument :start. Its initial
value is zero.

*ppp-default-end*  
[Variable]
This variable provides the default value for the keyword argument :end. Its initial
value is *number-of-processors-limit*, and it is reset to this value whenever a *cold-
boot is executed.
list-of-active-processors

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

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.
Chapter 6
Communication

CAUTION
The documentation in this chapter is largely obsolete. In each case, the nature of changes made with Version 5.0 are noted and references to auxiliary documentation are provided.

6.1 Communication between Processors

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 functions that perform communication are pref!!., the parallel version of pref, and *pset.

pref!! pvar-expression cube-address-pvar
&optional collision-mode

[Macro]

Note: The syntax and semantics of pref!! have been changed. The optional arguments have been changed into keyword arguments and a new :collision-mode value has been introduced. See chapter 6 of the Supplement to the *Lisp Reference Manual for more information.

pref!! returns 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 processors from which the data is
being retrieved. Unlike `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.

The `pref!!` macro takes an additional optional argument, `collision-mode`, that deter-
mines how cases are handled for efficiency where more than one processor is reading
from a single processor. The values allowed are `:collisions-allowed`, `:no-collisions`,
and `:many-collisions`. This argument allows *Lisp to optimize calls to `pref!!` in the
cases where each address is unique, as in `:no-collisions`, or when many addresses are
identical, as in `:many-collisions`.

`:collisions-allowed`
Each processor may access any other processor and multiple reads are al-
lowed. The time required to complete this operation is proportional to the
maximum number of processors reading from a single processor.

`:no-collisions`
This tells *Lisp that no two processors will ever be caused to read from the
same processor. It allows the Connection Machine to execute the read signifi-
cantly faster than the `:collisions-allowed` case does. However, if two proces-
sors do attempt to read from the same processor, the behavior is unpredict-
able. Use with caution!

`:many-collisions`
This is useful when there are many processors reading from a single processor.
*Lisp uses a different algorithm to resolve the collisions. The result is that the
`pref!!` almost takes constant time regardless of the routing pattern. That time is
approximately the same as a `pref!!` using `:collisions-allowed` where thirty proc-
essors are reading from a single processor, although this may vary for different
virtual processor ratios.

`pref-grid!! pvar-expression &rest grid-address-pvars`
&optional `collision-mode`
&key `:border-pvar` `border-pvar`

Note: This operation is obsolete.
The function \texttt{pref-grid!!} performs the same operation as \texttt{pref!!}; it simply allows grid addressing instead of cube addressing. It returns a \texttt{pvar} that contains the value of \texttt{pvar-expression} from the processors addressed by \texttt{grid-address-pvars}. This function evaluates \texttt{pvar-expression} differently from other \*Lisp operators; instead of evaluating \texttt{pvar-expression} in the currently selected set, it is evaluated in the context of the processors from which the data is being retrieved.

There must be as many \texttt{grid-address-pvars} as there are Connection Machine dimensions. Unlike \texttt{pvar-expression}, the \texttt{grid-address-pvars} are evaluated like normal expressions (i.e., in the processors of the currently selected set).

The optional argument \texttt{collision-mode} is used in the same way for \texttt{pref-grid!!} as for \texttt{pref!!}.

It is an error to read from a nonexistent processor. However, if the keyword \texttt{:border-pvar} is provided, and if the \texttt{grid-address-pvars} in a given processor \texttt{p} access a nonexistent processor, then the value of \texttt{border-pvar} in processor \texttt{p} is returned instead. During the evaluation of \texttt{border-pvar}, the currently selected set is set to only those processors reading off the edge of the Connection Machine grid. In order to better understand this behavior, consider the following two pieces of code:

\begin{verbatim}
(pref-grid!! source x-address y-address :border-pvar foo)

(if!! (off-grid-border-p!! x-address y-address)
     foo
     (pref-grid!! source x-address y-address))
\end{verbatim}

These are equivalent except the \texttt{pref-grid!!} ensures that \texttt{x-address} and \texttt{y-address} are evaluated exactly once each.

Again, if the value of \texttt{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.

\begin{verbatim}
(pref-grid-relative!! pvar-expression &rest relative-address-pvars
  &key :border-pvar border-pvar)
\end{verbatim}

\textbf{Note:} This operation is obsolete. It is replaced by the new operations \texttt{news!!} and \texttt{news-border!!}. See chapter 6 of the Supplement to the \*Lisp Reference Manual for more information.
This function behaves like `pref-grid!!`, except that relative addressing is used instead of absolute addressing, and there is no optional collision mode argument. An example of the use of this function is given later in this chapter.

This function is especially fast when the `relative-address-pvars` are all constants (as in `(! x)`).

As with the serial function `pref`, `*setf` may be applied to `pref!!`, `pref-grid!!`, and `pref-grid-relative!!` to write into memory instead of reading from it. In this case, `pvar-expression` is referred to as `dest-pvar` and must be a valid destination `pvar`.

When using `*setf` in this manner, `dest-pvar` is modified only in those processors that were accessed. Processors that were not written into retain the previous contents of `dest-pvar`. An error is signaled if a nonexistent 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 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`) signal an error. The function `*pset` allows multiple writes to combine in various ways without producing errors.

```
*pset combiner value-pvar dest-pvar cube-address-pvar
    &optional notify-pvar collision-mode

[Macro]
```

Note: The syntax of this macro has changed. The optional arguments are now keyword arguments and new keyword arguments have been added. See chapter 6 of the Supplement to the *Lisp Reference Manual for more information.

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

- `:default` — If the same address is written twice, an error is signaled. This is the same as using `setf` with `pref!!`.
- `:overwrite` — Only one write per address is successful. All other writes are discarded.
- `:or` — If two or more values are written into a single processor, the final value is the logical or of those values.
:and — If two or more values are written into a single processor, the final value is the logical and of those values.

:logior — If two or more values are written into a single processor, the final value is the bitwise or of those values. value-pvar must contain integers only.

:logand — If two or more values are written into a single processor, the final value is the bitwise and of those values. value-pvar must contain integers only.

:add — If two or more values are written into a single processor, the final value is the numerical sum of those values.

:max — If two or more values are written into a single processor, the final value is the numerical maximum of those values.

:min — If two or more values are written into a single processor, the final value is the numerical minimum of those values.

:no-collisions — This tells *Lisp that no two processors will ever be caused to read from the same processor. It allows the Connection Machine to execute the write somewhat faster than the other combiners. However, if two processors do attempt to write to the same processor, the behavior is unpredictable and the code is in error. Use with caution!

The :logior and :logand combiners are especially fast. The :or and :and are faster when the pvar being sent contains only t's and nil's.

The optional argument notify-pvar must be a pvar. Its value when *pset has finished executing is t in all processors into which a value is written, even if the value written happens to be the same as the pvar's current value, and is not affected in other processors.

The collision-mode argument is obsolete and no longer useful as of Version 5.0.

*pset-grid combiner value-pvar dest-pvar
&rest grid-address-pvars
&optional notify-pvar collision-mode

[Macro]

Note: This macro is obsolete.

This is analogous to *pset, except that the grid addressing is used.
*pset-grid-relative combiner value-pvar dest-pvar
&rest relative-grid-address-pvars

[Macro]

Note: This macro is obsolete.

This is analogous to *pset-grid, except that relative grid addressing is used and there are no optional notify-pvar and collision-mode arguments. This function is especially fast when the relative-address-pvars are all constants (as in (!! x)).

The following 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!!)))))

In the above examples, the form (*when (>!! (self-address!!) (!! 0)) prevents processor 0 from reading processor -1.

This function:

(*defun sum-a-pvar (pvar)
  (pref
    (*let (the-sum-goes-here)
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 four processors to its north, east, west and south.

6.2 Block Data Transfer between the Front End and the Connection Machine System

Transferring data between the front-end computer and the Connection Machine may be done much more efficiently when either the source or destination of the transfer is an array. Instead of repetitively calling pref, or *setf on pref, portions of the array can be moved in block mode using the functions described below.
**pvar-to-array** source-pvar &optional dest-array

&key (:array-offset 0)
  (:cube-address-start 0)
  (:cube-address-end *number-of-processors-limit*)

This function moves data from source-pvar into dest-array in cube-address order. If provided, dest-array must be one-dimensional. If a dest-array is not provided, an array is created of size :cube-address-end minus :cube-address-start. The data from source-pvar in processors :cube-address-start through 1 - :cube-address-end are written into the dest-array elements starting with element :array-offset. The result returned by pvar-to-array is dest-array.

**array-to-pvar** source-array &optional dest-pvar

&key (:array-offset 0)
  (:cube-address-start 0)
  (:cube-address-end *number-of-processors-limit*)

This function moves data from source-array to dest-pvar. The source-array must be one-dimensional. The other arguments behave the same way as in pvar-into-array. If a dest-pvar is not provided, array-to-pvar creates a destination pvar, in which case the call to the function must be within a function that accepts pvar expressions, such as *set. If a destination pvar is created, its value in processors to which array-to-pvar did not write is undefined. The value returned by this function is dest-pvar.

**pvar-to-array-grid** source-pvar &optional dest-array

&key (:array-offset (make-list *number-of-dimensions* :initial-element 0))
  (:grid-start (make-list *number-of-dimensions* :initial-element 0))
  (:grid-end *current-cm-configuration*)

This function moves data from source-pvar into dest-array in grid address order. If provided, dest-array must have the same number of dimensions as the current Connection Machine configuration. If dest-array is not specified, an array is created with dimensions :grid-end minus :grid-start, where the subtraction is done component-wise to produce a list suitable for make-array. The data from source-pvar in the sub-grid defined by :grid-start and :grid-end as the upper and lower corners, respectively, are written into a similar sub-grid of dest-array starting with element :array-offset as the upper corner. The arguments :array-offset, :grid-start, and :grid-end must be lists of length *number-of-dimensions*. The value returned by pvars-into-array-grid is dest-array.
array-to-pvar-grid source-array &optional dest-pvar &key (:array-offset
(make-list *number-of-dimensions*
  :initial-element 0))
(:grid-start
(make-list *number-of-dimensions*
  :initial-element 0))
(:grid-end *current-cm-configuration*))

This function moves data from source-array to dest-pvar in grid address order. The number of dimensions source-array has must be equal to *number-of-dimensions*. The other arguments to this function behave the same way as in pvar-to-array-grid. If dest-pvar is nil, array-to-pvar-grid creates a destination pvar, in which case the call to the function must be within a function that accepts pvar expressions, such as *set. If a destination pvar is created, its value in processors to which array-to-pvar-grid did not write is undefined. The value returned is dest-pvar.

### 6.3 Scan Functions

scan!! pvar function &key (:direction :forward)
:segment-pvar segment-pvar (:include-self t)

Note: In Version 5.0, scan!! has been given a :dimension keyword argument, rendering the function scan-grid!! obsolete. The function scan!! can now be used to scan across any axis of an n-dimensional grid. (Cf. the functions spread!! and reduce-and-spread!!), defined in chapter 6 of the Supplement to the *Lisp Reference Manual.)

For each selected processor, the value returned to that processor is the result of reducing the pvar values in all the processors preceding it. Its own pvar value is by default included in the reduction as well. “Reducing” in this context refers to the Common Lisp function reduce, which accepts two arguments, function and sequence. The reduce function applies function, which must be a binary associative function, to all the elements of the sequence. For example, if + were the function all the elements in sequence would be summed. In the case of a scan!! function, the sequence becomes the pvar values contained in the ordered set of selected processors.

The scan!! argument function is one of the following associative binary *Lisp functions: +!!, and!!, or!!, logand!!, logior!!, logxor!!, max!!, and min!!. In addition, the copy!! function is supported as a scanning function even though there is no such *Lisp function. In the following illustration, * is any of these binary functions:
<table>
<thead>
<tr>
<th>(self-address!!)</th>
<th>processor-selected?</th>
<th>value of pvar</th>
<th>result of scan</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>yes</td>
<td>c</td>
<td>b*c</td>
</tr>
<tr>
<td>3</td>
<td>no</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>yes</td>
<td>e</td>
<td>(b*c)*e</td>
</tr>
<tr>
<td>5</td>
<td>no</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>yes</td>
<td>g</td>
<td>((b*c)*e)*g</td>
</tr>
<tr>
<td>7</td>
<td>no</td>
<td>h</td>
<td></td>
</tr>
</tbody>
</table>

If * were the function +!!; this would be a summation over the set of selected processors, ordered by cube address:

```
(self-address!!) => 0 1 2 3 4 5 6 7...
(scan!! (self-address!!) '+!!)) => 0 1 3 6 10 15 21 28 ...
```

Normally, the value returned for the last selected processor is the result of applying the function to all the preceding selected processors and the last one. One may, however, break up the processors into segments. A segment consists of a sequence of processors in ascending cube-address order. A new segment of processors begins at each processor in which segment-pvar is non-nil. Even if all segment-pvar components are nil, however, there is always at least one segment beginning with the selected processor having the lowest cube address. The first processor in a segment always receives the value of pvar instead of the reduction of all the preceding processors. For example:

```
(self-address!!) => 0 1 2 3 4 5 6 7...
segment-pvar => nil nil nil t t nil nil t
(scan!! (self-address!!) '+!!)
 :segment-pvar segment-pvar) => 0 1 3 3 4 9 15 7...
```

In this example there are four segments. The first is 0, 1, 2; second is 3; third is 4, 5, 6; and fourth is 7... .

The amount of time required to execute a segmented scan is essentially fixed. That is, execution time does not vary significantly with the number or length of the segments or with the number of selected processors.

Unlike the other functions that can be used as arguments, copy!! exists only as a scanning function. It is used only in conjunction with segment-pvar. It causes the value of pvar in the first processor of a segment to be copied into all the other processors of that segment. For example:
Chapter 6. Communication

The direction of the scanning is normally from lowest to highest cube-address. If the `:direction` argument is `:backward`, then the scan is from highest to lowest cube-address. When scanning backward, segments are sequences of processors in descending cube-address order. In this example, the segments consist of, first, ...7, 6, 5; next, 4; and last, 3, 2, 1, 0.

Normally, each processor receives the result of applying `function` to all the processors before it up to and including itself. The `:include-self` keyword controls whether the value of a processor is included. When `:include-self` is `nil`, there are two effects:

1. The value of each processor is the result of applying `function` to all processors before it, excluding itself.
2. The value of processors in which `segment-pvar` is non-nil is the result of applying `function` to all the processors of the previous segment.

Following are two examples:

In this first example, the case where `:include-self` is `t` is identical to the first `scan!!` example. The processor of cube-address 3 receives its own address added to no others, being the first processor in a new segment. In the second case, where `:include-self` is `nil`, the value of processor 0 is undefined since there are no processors preceding it. Processor 3 receives a value that is the sum of the previous segment’s processor ad-

```
(self-address!!) => 0 1 2 3 4 5 6 7...
segment-pvar => nil nil nil t t nil nil t
(scan!! (self-address!!) 'copy!! :segment-pvar segment-pvar) => 0 0 3 4 4 4 7...
```

```
(self-address!!) => 0 1 2 3 4 5 6 7...
segment-pvar => nil nil nil t t nil nil t
(scan!! (self-address!!) '+!! :segment-pvar segment-pvar :include-self t) => 6 6 5 3 4 18 13 7...
```

```
(self-address!!) => 0 1 2 3 4 5 6 7...
segment-pvar => nil nil nil t t nil nil t
(scan!! (self-address!!) '+!! :segment-pvar segment-pvar :include-self nil) => * 0 1 3 3 4 9 15...
```
addresses, 0 + 1 + 2. Likewise, processor 7 receives a value that is the sum of the previous segment's processor addresses, 4 + 5 + 6.

The second example illustrates the double effect achieved when :include-self is nil, using the **max!!** function:

\[
\begin{align*}
\text{pvar} & \quad => 1 \ 10 \ 5 \ 20 \ 3 \ 4 \ 5 \ 6 \\
\text{segment-pvar} & \quad => \text{nil} \ \text{nil} \ \text{nil} \ \text{t} \ \text{t} \ \text{nil} \ \text{nil} \ \text{t} \\
(\text{scan!! pvar } ^\text{max!!} \\
\quad :\text{segment-pvar segment-pvar} \\
\quad :\text{include-self t}) & \quad => 1 \ 10 \ 10 \ 20 \ 3 \ 4 \ 5 \ 6 \\
(\text{scan!! pvar } ^\text{max!!} \\
\quad :\text{segment-pvar segment-pvar} \\
\quad :\text{include-self nil}) & \quad => * \ 1 \ 10 \ 10 \ 20 \ 3 \ 4 \ 5 \\
\end{align*}
\]

Scanning can be accomplished using grid addressing as well as cube addressing.

**scan-grid!! pvar function &key (:dimension :x) [Function]**

\[
(\text{direction :forward}) \\
\quad :\text{segment-pvar segment-pvar} \\
\quad (:\text{include-self t})
\]

**Note:** This function is obsolete. Its functionality is now included in that of **scan!!**.

The function **scan-grid!!** is similar to **scan!!** except that processors are scanned in grid order instead of cube order. The keyword argument :dimension controls whether the scanning is done across rows or columns. The value may be one of the symbols :x (rows) or :y (columns), or it may be a nonnegative integer less than **number-of-dimensions**. A :dimension value of 0 corresponds to rows and a value of 1 corresponds to columns.

**scan-grid!!** always performs a segmented scan, with each row or column beginning a new segment. If a :segment-pvar argument is supplied, then rows or columns are subsegmented according to the component values of **segment-pvar**. That is, a new subsegment begins at each non-nil value of **segment-pvar**.

As with **scan!!**, the amount of time it takes **scan-grid!!** to execute a segmented scan is essentially fixed. That is, execution time does not vary significantly with the number or length of the segments or with the number of selected processors.
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

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

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

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 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**  
[*Defun*]  
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.

### 6.5 Processor Addressing

This section contains functions that handle address generation and translation. When a dimension number is required, remember that it is always zero-based; in other words, the first dimension is dimension 0, the second dimension is dimension 1, and so on.

Note: New functions have been added that do inter-vp-set address generation and translation. See chapter 6 of the Supplement to the *Lisp Reference Manual, where the functions defined here are also redefined.

**self-address!!**  
[Function]  
This function returns a pvar that contains the cube address of each selected processor.

**self-address-grid!! dimension-pvar**  
[Function]  
This function 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**  
[Function]  
This function takes a cube-address and returns the grid address for the specified dimension. This function executes entirely in the front-end computer.

**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. This function executes entirely in the front-end computer.
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.
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
Machine hardware. All *defvar pvars are reallocated and their initial values are recomputed.

In addition, the user may specify the initial-dimensions of the Connection Machine processor configuration. This keyword argument is a list of dimension sizes. The dimensions must be powers of 2. If no initial-dimensions are specified, then they default to the same values as in the previous call to *cold-boot. If there was no previous call, the default is a two-dimensional grid.

*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.

Users may explicitly specify what hardware configuration to use by calling the Lisp/Paris function cm:attach before calling *cold-boot. This function is described in the Connection Machine Front-End Subsystems. *cold-boot calls cm:attach if the hardware is not already attached.

Some examples follow:

;;; This configures 64 x 128 processors, since that is the hardware configuration for an 8k machine.
(*cold-boot) ; 8k physical processors

;;; This configures 2 x 1 virtual processors per physical processor
;;; on a machine with 8k physical processors
(*cold-boot :initial-dimensions '(128 128)) ; 16k virtual processors

;;; This configures 1 virtual processor per physical processor
;;; because 128 x 128 is the physical size of a 16k machine.
(cm:attach :16k)
(*cold-boot :initial-dimensions '(128 128)) ; 16k physical processors

*warm-boot

This function must be called whenever a *Lisp program is abnormally terminated for any reason. The function resets 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 and aborted code may have left the Connection Machine hardware in an inconsistent state.
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.

\[
\text{add-initialization} \quad \text{name-of-form form variable} \quad [\text{Function}]
\]

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:

\[
(\text{add-initialization} (\text{string} \ '\text{items})
  \ '\text{initialization-items})
  \ '*after-*warm-boot-initializations*)
\]

\[
\text{delete-initialization} \quad \text{name-of-form variable} \quad [\text{Function}]
\]

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:

\[
(\text{delete-initialization} (\text{string} \ '\text{items})
  \ '*after-*warm-boot-initializations*)
\]
7.1.2 Configuration Variables

*Lisp provides a number of variables whose values are set by *cold-boot and *with-vp-set (new with Version 5.0). A program using these configuration variables will run in any configuration. The user must not modify the values of any of these configuration variables.

Note: Several new configuration variables have been defined with Version 5.0. See chapter 5 of the Supplement to the *Lisp Reference Manual.

*number-of-processors-limit* [Variable]
This variable specifies the effective number of processors a user program sees. For a machine with 65,536 physical processors, each simulating 16 processors, this variable contains 1,048,576.

*log-number-of-processors-limit* [Variable]
This variable provides the logarithm, base 2, of the number of processors available.

*number-of-dimensions* [Variable]
This variable is defined when *cold-boot is run. Its value is the number of dimensions given. The default value is 2. The current interpreter supports only two dimensions.

*current-cm-configuration* [Variable]
This variable is a list containing each dimension's size in the current configuration.

dimension-size dimension [Function]
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.

7.1.3 The *Lisp Simulator

The *Lisp simulator, which runs on the serial front end, strives to be an exact simulation of the *Lisp interpreter running on the Connection Machine. The functions *cold-
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.
(pvar-mantissa-length pvar)
(pvar-exponent-length pvar)

If `pvar` is of type `defined-float` or `complex`, then these functions return the mantissa and exponent lengths.

Following is an example of *Lisp code with embedded Paris instructions:

```lisp
(*let ((cube-address (self-address!!))
   dest
   (source (!! 100)))
(declare (type (pvar (unsigned-byte cm:*cube-address-length*))
   cube-address)
   (type (pvar (signed-byte 10) dest source)))
(cm:send (pvar-location dest) (pvar-location cube-address)
   (pvar-location source) (pvar-length source)))
```

### 7.3 *Lisp Memory Management:
#### Stack and Heap Storage

The memory of each Connection Machine processor is broken up into a stack, a heap, and a gap, along with a small number of reserved bits. The space occupied by these blocks is identical in all Connection Machine processors and is described by several variables in the front-end computer.

The stack begins near the low end of Connection Machine memory. It grows as necessary into the gap, but may not extend past the beginning of the heap.

The stack is used for fast allocation of pvars returned as results of Lisp and *Lisp functions (such as `+!!`), and for storage of all pvars created by `*let`. As with a standard stack, memory is allocated and deallocated on a first in, last out basis.

The heap is used to store all other pvars that may not be allocated and deallocated in the strict order required by a stack. This includes all pvars created by `*defvar` and `allocate!!`. 
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)))))
```
Since the body of the *let defining xyzzy has been exited at the time the lambda-defined function is actually called, the *Lisp language definition makes no guarantee that the pvar xyzzy still contains constant anywhere.

Note that *let and *let* allow one to return a pvar created by the *let as the value of the *let and use that returned value. It is only attempting to access a value created by *let within a lexical closure that is doomed to failure.

8.3 Using Mapcar on Functions that Return pvars

Consider the following code:

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

pvar-list is a list of pvars that have been allocated on the *Lisp stack. The *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 predefined-pvar-1 and predefined-pvar-2 will contain the same values.

8.4 *warm-boot

Whenever a *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 *Lisp code again. This is because both the currently selected set and certain internal *Lisp variables are left in an inconsistent state. *warm-boot clears all Connection Machine error conditions (hardware and microcode) without modifying the contents of the Connection Machine memory. Very bizarre behavior results when this dictate is not followed.

*warm-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.
8.5 *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:

```lisp
(*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

```lisp
(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.
8.8 Grid and Cube Addresses

Programmers should not write code that depends on a particular mapping between grid and cube addresses, as this may change in future Connection Machine implementations. Instead, use the address-translation functions provided.
Appendix
Appendix A

*Lisp Symbols

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

Constants

nil!!, 7, 25
t!!, 7, 25

Operators

!!, 28
+!!, 28
-!! , 28
*!! , 28
*apply, 33
*cold-boot, 10
*defun, 9, 33, 34
*funcall, 33
*pref!!, 10
*set, 7, 8
*when, 7
/!! , 28
/=!! , 25
=!! , 24
>!! , 25
>=!! , 25
1+!! , 28
1-!! , 28
add-initialization, 55
*all, 19
allocate!!, 11, 12
*and, 49
and!!, 26
array-to-pvar, 44
array-to-pvar-grid, 45
ash!!, 29
ceiling!!, 29
*cold-boot, 53, 61
*cond, 20
cond!! , 32
cos!!, 30
cube-from-grid-address, 50
cube-from-grid-address!!, 51
*deallocate, 12
*deallocate-*defvars, 12
declare, 16
*defun, 15
delete-initialization, 55
deposit-byte!!, 31
do-for-selected-processors, 20
enumerate!!, 32
eq!!, 24
eq!!!, 24
evenp!!, 23
float!!, 30
floatp!!, 24
floor!!, 29
grid-from-cube-address, 50
grid-from-cube-address!!, 51
*if, 20
if!!, 31
integerp!!, 24
isqrt!!, 29
*let, 13, 15
*let*, 13, 15
list-of-active-processors, 35
load-byte!!, 30
log!!, 30
*logand, 49
logand!!, 27
logeqv!!, 27
*logior, 49
logior!!, 27
lognot!!, 27
logxor!!, 27
*max, 49
max!!, 28
*min, 49
min!!, 28
minusp!!, 24
mod!!, 29
not!!, 26
numberp!!, 24
odd!!, 23
off-grid-border-p!!, 51
off-grid-border-relative-p!!, 51
*or, 49
or!!, 26
plusp!!, 23
pref, 14
pref!!, 37, 61
pref-grid, 14
pref-grid!!, 38, 61
pref-grid-relative!!, 39, 61
pretty-print-pvar-in-currently
selected-set, 35
*proclaim, 15
*pset, 40
*pset-grid, 41
*pset-grid-relative, 42
pvar-to-array, 44
pvar-to-array-grid, 44
pvarp, 12
random!!, 29
rank!!, 32
rotl!!, 30
round!!, 29
sert!!, 45
scan–grid!!, 48
self-address, 7
self-address!!, 50
self-address–grid!!, 50
*set, 14
signum!!, 30
sin!!, 30
sqrt!!, 29
*sum, 50
the, 15
truncate!!, 29
type, 16
*warm-boot, 54, 60
*when, 19
with-css-saved, 20
xor!!, 26
zerop!!, 24

Pvar Types
double-float, 16
long-float, 16
unsigned-byte, 16
pref, 6
short-float, 16
signed-byte, 16
single-float, 16

Variables
*after–*cold-boot-initializations*, 55
*after–*warm-boot-initializations*, 55
*before–*cold-boot-initializations*, 55
*before–*warm-boot-initializations*, 55
*current-cm-configuration*, 56
dimension-size, 56
*log-number-of-processors-limit*, 56
*number-of-dimensions*, 56
*number-of-processors-limit*, 56
*ppp-default-end*, 34
*ppp-default-format*, 34
*ppp-default-mode*, 34
*ppp-default-per-line*, 34
*ppp-default-start*, 34
## Index

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

<table>
<thead>
<tr>
<th>C</th>
<th>multiple values, 61</th>
</tr>
</thead>
<tbody>
<tr>
<td>communication, interprocessor, 9</td>
<td></td>
</tr>
<tr>
<td>cube address, 4, 62</td>
<td></td>
</tr>
<tr>
<td>currently selected set, 5, 19</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>debugging tools, 33</td>
<td></td>
</tr>
<tr>
<td>declare statement, 16</td>
<td></td>
</tr>
<tr>
<td>defining *Lisp functions, 9</td>
<td></td>
</tr>
<tr>
<td>*defvar, 11</td>
<td></td>
</tr>
<tr>
<td>double-float pvar, 16</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>field, 5</td>
<td></td>
</tr>
<tr>
<td>front-end computer, data transfer, 43</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>G</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>grid address, 4, 62</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>interprocessor, 37—51</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>logical operations, 26—27</td>
<td></td>
</tr>
<tr>
<td>long-float pvar, 16</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>memory management, 58</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>news!!, 10</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Paris, called from *Lisp, 57</td>
<td></td>
</tr>
<tr>
<td>predicate operations, 23</td>
<td></td>
</tr>
<tr>
<td>processor selection, 7</td>
<td></td>
</tr>
<tr>
<td>processors, 4</td>
<td></td>
</tr>
<tr>
<td>non-selected, 59</td>
<td></td>
</tr>
<tr>
<td>pvar, 4, 6, 11—17</td>
<td></td>
</tr>
<tr>
<td>pvar type declaration, 15—17</td>
<td></td>
</tr>
<tr>
<td>pvars, extent, 59</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>selection, of processors, 19—21</td>
<td></td>
</tr>
<tr>
<td>*set, 7</td>
<td></td>
</tr>
<tr>
<td>*setf, 6, 14</td>
<td></td>
</tr>
<tr>
<td>short-float pvar, 16</td>
<td></td>
</tr>
<tr>
<td>signed-byte pvar, 16</td>
<td></td>
</tr>
<tr>
<td>simulator, 56</td>
<td></td>
</tr>
<tr>
<td>single-float pvar, 16</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>type statement, 16</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>U</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned-byte pvar, 16</td>
<td></td>
</tr>
<tr>
<td>user-defined operations, 32</td>
<td></td>
</tr>
</tbody>
</table>