The
Connection Machine
System

Supplement to the *Lisp Reference Manual

Version 5.0
September 1988

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, September 1988

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142-1214
(617) 876-1111

Contents

Preface

Objectives of This Manual Supplement

The *Supplement to the *Lisp Reference Manual* provides reference information about new features added to the *Lisp language for the release of Version 5.0. It does *not* replace the ***Lisp Reference Manual***, Version 5.0.

Intended Audience

The reader is assumed to have a working knowledge of Common Lisp, as described in *Common Lisp: The Language*, and of *Lisp, as described in the ***Lisp Reference Manual***, Version 4.0. The reader is further assumed to have a general understanding of the Connection Machine system. The *Connection Machine Front-End Subsystems* manual provides the necessary background information about the Connection Machine system.

Revision Information

This supplement is new with *Lisp, Version 5.0.

Organization of This Manual

Chapter 1 Complex Number Pvars
The first chapter describes the definition and use of complex number pvars

Chapter 2 Character Pvars
The second chapter describes the definition and use of character pvars.

Chapter 3 Array Pvars
The third chapter describes the definition and use of array pvars.

Chapter 4 Structure Pvars
The fourth chapter describes the definition and use of structure pvars.

Chapter 5 Virtual Processor Sets
The fifth chapter explains the new virtual processor mechanism whereby multiple virtual processor configurations may be employed during a single session. Operations for defining and using virtual processor sets are described.

Chapter 6 N-Dimensional Interprocessor Communication
The sixth chapter explains how the new capability of defining *n*-dimensional virtual processor configurations affects communication within the Connection Machine. New *n*-dimensional communication facilities are described.

Chapter 7 Miscellaneous New *Lisp Operations
The seventh chapter provides reference information about a variety of new features introduced with version 5.0.

Chapter 8 .Parallel Variable Types
The eighth chapter describes all the valid pvar type specifiers supported by *Lisp and explains the rules of type conversion and coercion for each type.

Chapter 9 Scanning with Segment Sets
The ninth chapter describes an experimental feature that supports non-contiguous segmented scan operations.

Chapter 10 Parallel Vector Operations
The tenth chapter describes experimental operations that provide optimized manipulation of parallel vectors. Serial equivalents of many of these new operations are also described.

Chapter 11 Parallel Sequence Operations
The eleventh chapter describes experimental parallel equivalents of the Common Lisp sequence operations.

Appendix A The Relationship between the CM-2 Architecture, Paris, and *Lisp
The first appendix describes the CM-2 hardware capabilities accessible from *Lisp.

Appendix B Example Program 1: Text Processing
The second appendix is a sample program that demonstrates the use of several *Lisp features new with Version 5.0.

Appendix C Example Program 2: Determinants
The third appendix is a sample program that demonstrates the use of several *Lisp features new with Version 5.0.

Associated Documents

The following documents should be read in the order listed before reading the *Supplement to the *Lisp Reference Manual.*

- *Connection Machine Front-End Subsystems*

  This volume explains how to configure the Connection Machine system and how to access it from either a Symbolics Lisp Machine or a UNIX system. It includes:

  - *System Front Ends Release Notes*, Version 5.0
  - *CM User's Guide: UNIX System Front End*, Version 4.0
  - *CM User's Guide: Lisp System Front End*, Version 4.0

  Those working on a UNIX system front end should read both User's Guides; those working on a Symbolics Lisp Machine front end need only read the second.

- *Common Lisp: The Language* by Guy L. Steele Jr. Burlington Mass.: Digital Press, 1984

  This book defines the *de facto* industry standard Common Lisp.

- The **Lisp Reference Manual*, Version 5.0

  This manual provides a complete description of the *Lisp language through Version 4.0 and has been updated for Version 5.0. It covers the essential concepts of *Lisp and is supplemented by the *Supplement to the *Lisp Reference Manual*, Version 5.0

The following related documents should be read along with the *Supplement to the *Lisp Reference Manual.*

- The **Lisp Release Notes*, Version 5.0

  These release notes supersede all previous *Lisp release notes. They provide an overview of all changes made to the language, to the interpreter, and to the compiler for the release of version 5.0

- The **Lisp Compiler Guide*, Version 5.0

  This manual describes how to use the *Lisp compiler and provides helpful suggestions for writing *Lisp code that will compile.

The following documents are recommended.

- *Model CM-2 Technical Summary*

  This publication offers a succinct overview of the Connection Machine system.

- *Connection Machine Parallel Instruction Set*

  This volume describes Paris, the Connection Machine system assembly level programming language.

Notation Conventions

The notation and typograghical conventions used in this manual are reviewed below. These conventions closely follow—but are not identical to—those used in *Common Lisp: The Language*.

The symbol => indicates evaluation. The symbol -> indicates macro expansion.

Symbol names within text appear in bold modern style typeface, as in ***max**.

Code examples are set in typewriter style typeface, as in:

```
(cons abra cadabra) => (abra cadabra)
```

Metavariables, names that stand for pieces of code, appear in italics. For example, the names of arguments in function or macro descriptions appear in italics, as shown in the function description format below.

Function descriptions are presented as show below:

function-name!! *required-arg1-integer-pvar required-arg2-integer-pvar* [*Function*]
&optional *optional-arg-float-pvar optional-arg-char-pvar*
&rest *rest-pvars*
&key :key1 :key2 :keywest
&aux *aux-arg-vars*

In this example, the function **function-name!!** takes two required pvar arguments, *required-arg1-integer-pvar* and *required-arg2-integer-pvar*. Required arguments are always shown immediately after the function name in a function description. If present, optional arguments are preceded by the appropriate lambda-list keywords: **&optional**, **&rest**, **&key**, and **&aux**.

The metavariable names used to represent arguments in function and macro descriptions indicate restrictions on argument type. Argument names with 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.

Plural metavariable names are used to indicate multiple optional arguments of the same type. The use of *rest-pvars* above demonstrates this. If restrictions on order and type exist for optional arguments, these are reflected in the metavariable names. The metavariables *optional-arg-float-pvar* and *optional-arg-char-pvar* above are examples.

Keyword argument names only are specified in function descriptions. Allowable keyword values are enumerated and described in the text. Italicized metavariables are often used in the text to refer to the values of keyword arguments. For example, the value of the keyword **:keywest** would be referred to as *keywest*, and *keywest* might be restricted to symbols representing months of the year. Calling **function-name!!** with **:keywest 'February** would put you on the beach in winter.

Customer Support

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

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

To contact Thinking Machines Customer Support:

U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142-1214
Internet Electronic Mail:	customer-support@think.com
Usenet Electronic Mail:	harvard!think!customer-support
Telephone:	(617) 876-1111

For Symbolics users only:

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

```
To:  bug-connection-machine@think.com
```

Please supplement the automatic report with any further pertinent information.

Chapter 1

Complex Number Pvars

*Lisp version 5.0 implements pvars containing complex numbers. Parallel equivalents of most Common Lisp operations that accept complex numbers are now available in *Lisp. *Lisp imposes greater restrictions on some of these operations than does Common Lisp. These are fully described in the following discussion.

1.1 Complex Pvar Type Definition, Predication, and Coercion

As in Common Lisp, the real and imaginary components of complex numbers in *Lisp must each contain exactly the same data type. Unlike Common Lisp, complex pvars in *Lisp are restricted to having floating-point values in their real and imaginary components. The type declaration for a complex pvar includes, either implicitly or explicitly, a precision specification for these floating-point values.

The following shorthand type definitions are provided to allow the definition of complex pvars containing IEEE standard floating-point format components.

(pvar (complex single-float)) [*Type*]
(pvar (complex double-float)) [*Type*]

These forms specify single- and double-precision storage for both the real and imaginary components of complex number pvars.

A single-precision complex pvar uses 23 bits for the significand and 8 bits for the exponent of each component. A double-precision complex pvar uses 52 bits for the significand and 11 bits for the exponent of each component.

The complex pvar type is more explicitly specified as follows:

(pvar (complex (defined-float *significand exponent*))) [*Type*]

The *significand* and *exponent* specifiers determine the number of bits used to store the significand and exponent portions of the real floating-point numbers used for both the real and imaginary portions of a complex pvar.

Example:

```
(*let* (c1 c2)
  (declare (type (pvar (complex single-float)) c1))
  (declare (type (pvar (complex (defined-float 30 9))) c2))
  (*set c1 (sqrt!! (!! #C(-1.0 0.0))))
  (*set c2 (log!! (!! #C(-1.0 0.0))))
)
```

PERFORMANCE NOTE

On a CM-2 with the special floating-point accelerator, *Lisp code that uses complex pvars of type **(pvar (complex single-float))** in numeric calculations executes significantly faster than code that uses other types of complex pvars.

complexp!! *pvar* [*Function*]

This predicate returns **t** in each processor whose value of *numeric-pvar* is a complex number; it returns **nil** elsewhere. The argument *pvar* may be any pvar.

complex!! *realpart-pvar* **&optional** *imagpart-pvar* [*Function*]

This function returns a complex pvar that has, in each processor, the *realpart-pvar* component as its real part and the *imagpart-pvar* component as its imaginary part. Conversion according to the rule of floating-point contagion takes place as necessary. That is, the bit field lengths of the exponent and significand components of floating-point numbers in all active processors are guaranteed to be as large as the largest representation of either component in any active processor.

The arguments *realpart-pvar* and *imagpart-pvar* must be non-complex numeric pvars. If *imagpart* is not specified, then an imaginary part pvar of **(!! 0)** is provided.

```
(complex!! realpart-pvar)
<=>
(coerce!! realpart-pvar '(pvar (complex float))
```

1.1.1 Rules of Complex Contagion and Canonicalization

*Lisp does adhere to the rule of complex contagion as stated in *Common Lisp: The Language*. When an operation is passed a pvar that contains a mixture of non-complex and complex number components, the non-complex components are converted to complex numbers by providing an imaginary part of 0.0.

*Lisp does *not* adhere to the rule of complex canonicalization as stated in *Common Lisp: The Language*. That is, if the result of an operation on a complex pvar is, in any processor, a complex rational with a zero imaginary part that result is *not* converted to a non-complex rational consisting only of the real part: the zero imaginary part is preserved.

In Common Lisp

```
(complex 0) => #C(0 0) => 0
```

Given a complex number with a zero imaginary, Common Lisp drops the imaginary part. In contrast, in *Lisp

```
(complex!! (!! 0)) => (!! #C(0.0 0.0))
```

Notice that *Lisp coerces both parts of the resulting complex pvar into a floating-point representation.

1.2 Mathematical Operations on Complex Pvars

*Lisp provides parallel equivalents to most mathematical Common Lisp functions defined to accept complex arguments.

The following *Lisp functions accept complex arguments.

zerop!!	**numberp!!**
=!!	**abs!!**
/=!!	**phase!!**
+!!	**signum!!**
-!!	**conjugate!!**
***!!**	**realpart!!**
/!!	**imagpart!!**
1+!!	**exp!!**
1-!!	**expt!!**
***sum**	**log!!**
abs!!	**sqrt!!**

sin!!	**asin!!**	**sinh!!**	**asinh!!**
cos!!	**acos!!**	**cosh!!**	**acosh!!**
tan!!	**tanh!!**	**atan!!**	**atanh!!**
cis!!			

Most of these behave, in each processor, exactly like their Common Lisp counterparts. Those functions listed and underlined are not exact parallel equivalents of Common Lisp functions. To find detailed documentation of any of these operations, consult the Master Index.

1.2.1 Irrational and Transcendental Functions

*Lisp restricts the use of some irrational and transcendental functions with respect to complex numbers more strictly than does Common Lisp.

Complex Results

In *Lisp, it is an error if one of the following functions would have to return a complex number pvar when given floating-point pvar arguments:

sqrt!!	
expt!!	**log!!**
asin!!	**acos!!**
acosh!!	**atanh!!**

To get a complex result from **sqrt!!**, **asin!!**, **acos!!**, **acosh!!**, or **atanh!!**, it is necessary to first coerce its single argument into a complex pvar. To get a complex result from **expt!!** or **log!!**, it is necessary to first coerce only the first argument into a complex pvar. The

required coercion may be achieved by using the function **complex!!** or the function **coerce!!**.

For example:

```
(sqrt!! (!! -1)) => error
(sqrt!! (complex!! (!! -1))) => (complex!! (!! 0.0) (!! 1.0))

(expt!! (!! -1) (!! 0.5)) => error
(expt!! (complex!! (!! -1)) (!! 0.5))
=> (complex!! (!! 0.0) (!! 1.0))
```

Argument Restrictions

*Lisp restricts the values of arguments supplied to these functions as described below.

sqrt!! *numeric-pvar* [*Function*]

The non-negative square root of *numeric-pvar* is returned.

It is an error if the argument *numeric-pvar* is either a floating-point pvar that contains negative numbers or an integer pvar that contains negative numbers. The function **sqrt!!** will never return a complex pvar as its result unless *numeric-pvar* is complex.

expt!! *base-pvar power-pvar* [*Function*]

This function computes and returns a pvar containing *base-pvar* raised to the power *power-pvar* in each processor.

It is an error if the argument *base-pvar* is a negative floating-point pvar and the argument *power-pvar* is a floating-point pvar. It is also an error if the argument *base-pvar* is an integer pvar and argument *power-pvar* contains negative integers.

log!! *numeric-pvar* **&optional** *base* [*Function*]

The logarithm of *numeric-pvar* in base *base* is returned. If *base* is not supplied, the natural logarithm is returned.

The argument *numeric-pvar* must be either a non-negative floating-point pvar or a non-negative integer pvar. The argument *base* must be a positive, non-complex number pvar.

asin!! *numeric-pvar* [*Function*]
acos!! *numeric-pvar* [*Function*]
atanh!! *numeric-pvar* [*Function*]

These functions compute and return the arc sine, arc cosine, and the hyperbolic arc tangent of *numeric-pvar*, respectively.

It is an error if the argument *numeric-pvar* is a floating-point pvar or an integer pvar containing numbers of magnitude greater than 1.0.

acosh!! *numeric-pvar* [*Function*]

These functions compute and return the hyperbolic arc cosine and the hyperbolic arc tangent of *numeric-pvar*.

It is an error if the argument *numeric-pvar* is a floating-point pvar or an integer pvar containing numbers less than 1.0.

Chapter 2

Character Pvars

*Lisp Version 5.0 implements pvars containing characters. Parallel equivalents of almost all Common Lisp operations that accept character data are now available in *Lisp.

While arrays of characters are allowed in *Lisp, the parallel equivalent of strings is not provided by *Lisp. This follows from the restriction that array pvars of varying lengths in different processors are not supported in *Lisp.

2.1 Character Pvar Type Definition

There are two *Lisp pvar types that store character data. These are parallel equivalents of the Common Lisp character and string-char types.

(pvar character) [*Type*]
(pvar string-char) [*Type*]

Example:

```
(*let (ch1)
    (declare (type (pvar string-char) ch1))
    (*if (evenp!! (self-address!!))
        (*set ch1 (!! #\Q))
        (*set ch1 (!! #\L))
    )))
```

2.2 *Lisp Global Character Variables

In *Lisp, as in Common Lisp, character pvars have three attributes represented by three bit fields: the code, the bits, and the font fields. *Lisp provides variables that define the lengths of these fields as well as variables that define the upper bounds on the values these fields may contain.

***char-code-length** [*Variable*]

This defines the length in bits of the code subfield of a pvar character. The default is 8 bits. Pvars of type **(pvar string-char)** have only a code field and are the same length as ***char-code-length.**

***char-code-limit** [*Variable*]

This is the upper exclusive bound restricting the value of the pvar character code attribute. The default is 256.

***char-bits-length** [*Variable*]

This defines the length in bits of the bits subfield of a pvar character. The default is 4 bits.

***char-bits-limit** [*Variable*]

This is the upper exclusive bound restricting the value of the pvar character bits attribute. The default is 16.

***char-font-length** [*Variable*]

This defines the length in bits of the font subfield of a pvar character. The default is 4 bits.

***char-font-limit** [*Variable*]

This is the upper exclusive bound restricting the value of the pvar character font attribute. The default is 16.

***character-length** [*Variable*]

This defines the total length in bits of a pvar of type pvar character. The default is 16 bits.

2.2.1 Setting the Global Character Variables

initialize-character &key :code :bits :font :front-end-p :constantp [*Function*]

This function sets the values of the *Lisp character attributes, which are stored in global character variables. The **initialize-character** function should be called before ***cold-boot** is invoked.

A successful call to **initialize-character** returns zero values.

It is not necessary to call **initialize-character** unless *Lisp application code requires global character variable values that differ from the defaults. If this is necessary, **initialize-character** must be invoked before ***cold-boot**. Calling **initialize-character** during a session will cause existing character data to be garbled or lost.

The keywords **:code**, **:bits**, and **:font** take integer values specifying how many bits will be allocated for each attribute of any character pvar. The defaults are **:code** 8, **:bits** 4, and **:font** 4.

The value for **:code** must be greater than or equal to 7.

The value for **:bits** must be greater than 0.

The value for **:font** must be greater than or equal to 0.

The keyword **:front-end-p** takes either t or nil as a value and defaults to **nil**. If *front-end* is **t**, the global character variables are set to match the character storage format of the front end.

Symbolics front ends have their code, bits, and font lengths set to 16, 4, and 0, respectively. Under Lucid Common Lisp, these values are are 8, 4, and 0. **Note:** These front-end character attribute lengths are independent of the character attribute lengths on the Connection Machine system.

If code, bits, or font attributes are specified that differ in storage size from those of front-end scalar character data, then it is impossible for some characters created on the front end to be represented on the Connection Machine system or for some ele-

ments of character pvars to be represented on the front end. For example, with a UNIX front end running Lucid Common Lisp,

```
(initialize-character :code 9)
(*defvar foo (code-char!! (!! 511)))
(pref foo 0)
```

is in error. Given 9 code bits, the Connection Machine ***char-code-limit** becomes 512. Meanwhile, the front end has only 8 code bits and a **char-code-limit** of 256. Thus, the front end cannot represent the characters stored in **foo** because the character code value is too large.

The keyword **:constantp** takes a boolean value. This is used to specify whether or not the sizes of character attributes are consistent across sessions. The *Lisp compiler uses this distinction to choose between producing compiled code that uses the global character variables and producing compiled code that substitutes hard coded values for these variables. Therefore, code compiled with **:constantp t** will run reliably only in worlds where the character attributes are the size specified at compile-time. Code compiled with **:constantp nil**, need not be recompiled to move between worlds with different character attribute sizes.

2.3 Functions Operating on Character Pvars

*Lisp operations on character pvars are parallel equivalents of the Common Lisp character operations specified in chapter 13 of *Common Lisp: The Language*.

2.3.1 Functions to Access Character Attributes

char-code!! *character-pvar* [*Function*]
char-bits!! *character-pvar* [*Function*]
char-font!! *character-pvar* [*Function*]

These functions each return a pvar that contains the code, bits, or font attributes of each character element of *character-pvar*.

The argument *character-pvar* must be a character pvar, a string-char pvar, or a general pvar containing only character or string-char elements.

By definition, the font and bits attributes of a string-char pvar are zero. It is always the case that:

```
(char-bits!! string-char-pvar) => (!! 0)
(char-font!! string-char-pvar) => (!! 0)
```

2.3.2 Functions to Construct and Convert Characters

code-char!! *code-pvar* **&optional** *bits-pvar font-pvar* [*Function*]

This function attempts to construct a character pvar with the specified attributes. In processors where this can be done, the resulting character is returned. In processors where this can not be done, **nil** is returned.

All three arguments must be non-negative integer pvars. The optional *bits-pvar* argument and the optional *font-pvar* argument each default to **(!! 0)**.

make-char!! *character-pvar* **&optional** *bits-pvar font-pvar* [*Function*]

This function attempts to construct a character pvar with the same code attribute as *character-pvar* and with the optionally specified bits and font attributes. In processors where this can be done, the resulting character is returned. In processors where this can not be done, **nil** is returned.

The argument *character-pvar* must be a character pvar, a string-char pvar, or a general pvar containing only character or string-char elements.

Both optional arguments must be non-negative integer pvars; each defaults to **(!! 0)**.

character!! *char-or-int-pvar* [*Function*]

Type coercion is attempted on the argument *char-or-int-pvar*. In processors where this is successful, the resulting character is returned. In processors where this is unsuccessful, **character!!** returns **nil**.

The argument *char-or-int-pvar* must be a pvar of type character, string-char, integer, or a general pvar containing only elements of these types.

```
(character!! char-or-int-pvar)
<=>
(coerce!! char-or-int-pvar '(pvar character))
```

char-upcase!! *character-pvar* [*Function*]
char-downcase!! *character-pvar* [*Function*]
char-flipcase!! *character-pvar* [*Function*]

These functions attempt to convert the case of each character element of *character-pvar*. The return value of either operation is a pvar containing converted characters where possible and intact original character values elsewhere. During these case conversions, the values of the bits and font attributes are not changed. Notice that only alphabetic characters are susceptible to case conversion. Thus, characters with non-zero bit field values will not be changed.

The argument *character-pvar* must be a pvar of type character or string-char, or a general pvar containing only elements of these types.

digit-char!! *weight-pvar* **&optional** *radix-pvar font-pvar* [*Function*]

This function attempts to construct a character pvar containing, in each processor, a character of font *font-pvar* such that, taken as a digit of radix *radix-pvar*, that character has weight *weight-pvar*. In each processor where this is possible, the resulting character is returned. In each processor where this is not possible, **nil** is returned.

All arguments must be non-negative integer pvars.

The function **digit-char!!** will never return **nil** in a processor where the value of *font-pvar* is 0, that of radix-*pvar* is between 2 and 36 inclusive, and that of *weight-pvar* is less than *radix-pvar*.

If a character having both upper and lower case representations will satisfy **digit-char!!**, upper case letters are preferred. For example,

```
(digit-char!! (!! 14) (!! 16) ) => (!! #\E)
```

char-int!! *character-pvar* [*Function*]

This function translates a character pvar into an integer pvar.

The return value is a non-negative integer pvar that holds the implementation-dependent encoding of each character in *character-pvar*.

The argument *character-pvar* must be a pvar of type character or string-char, or a general pvar containing only elements of these types.

The **char-int!!** function relies on the Connection Machine system's encoding of characters. Results obtained from this function should not be expected to conform to results obtained from the Common Lisp function **char-int** run on front-end machines.

int-char!! *integer-pvar* [*Function*]

This function is the converse of **char-int!!**. It converts an integer pvar into a character pvar. The return value is a character pvar which, if given to **char-int!!**, will return *integer-pvar*.

The argument *integer-pvar* must be a non-negative integer pvar.

The **int-char!!** function relies on the Connection Machine system's encoding of characters. Results obtained from this function should not be expected to conform to results obtained from the Common Lisp function **int-char** run on front-end machines.

2.3.3 Character Predicate Tests

Each of the following functions tests its argument and returns a boolean pvar.

characterp!! *pvar* [*Function*]

This function returns **t** in those processors where character-pvar contains character data and **nil** elsewhere.

The argument pvar may be any pvar.

string-char-p!! *character-pvar* [*Function*]

This function returns **t** in those processors where character-pvar contains string-char data and **nil** in processors where *character-pvar* contains character data. To pass this string-char type test, an element of *character-pvar* must have bits and font attributes that are each of zero value.

The argument *character-pvar* must be a character pvar, a string-char pvar, or a general pvar containing only elements of type character or string-char.

standard-char-p!! *character-pvar* [*Function*]

This function returns **t** in those processors where *character-pvar* contains an element of type **standard-char**; it returns **nil** elsewhere. The Common Lisp definition of **standard-char** is used. To pass this type test, the value of *character-pvar*'s bits and font attributes must both be zero.

The argument *character-pvar* must be a character pvar, a string-char pvar, or a general pvar containing only elements of type character or string-char.

graphic-char-p!! *character-pvar* [*Function*]

This function returns **t** in those processors where *character-pvar* contains a printing character and **nil** elsewhere. On the Connection Machine, only characters with ASCII values ranging from 32 to 127, inclusive, are considered graphic, printing characters. Any character pvar with a bits field of non-zero value is not a graphic character pvar.

The argument *character-pvar* must be a character pvar, a string-char pvar, or a general pvar containing only elements of type character or string-char.

alpha-char-p!! *character-pvar* [*Function*]

This function tests its argument for alphabetic elements. In those processors where *character-pvar* contains an alphabetic element, **t** is returned. In those processors where *character-pvar* does not contain an alphabetic element, nil is returned.

The argument *character-pvar* must be a character pvar, a string-char pvar, or a general pvar containing only elements of type character or string-char.

upper-case-p!! *character-pvar* [*Function*]
lower-case-p!! *character-pvar* [*Function*]
both-case-p!! *character-pvar* [*Function*]

These predicates test the case of the character components of *character-pvar*.

The argument *character-pvar* must be a character pvar, a string-char pvar, or a general pvar containing only elements of type character or string-char.

Where *character-pvar* contains characters in the range A through Z, **upper-case-p!!** returns **t**.

Where *character-pvar* contains characters in the range a through z, **lower-case-p!!** returns **t**.

Where *character-pvar* contains characters which, regardless of current case, may be represented in both upper and lower case, **both-case-p!!** returns **t**.

For each function, the return value is **nil** in those processors containing character data that fails to pass the test criterion.

digit-char-p!! *character-pvar* **&optional** *radix-pvar* [*Function*]

This function tests *character-pvar* for digits of radix *radix-pvar*.

In each processor containing a *character-pvar* element that is a digit of the specified radix, **digit-char-p!!** returns a non-negative integer indicating the weight of the digit. In those processors where the elements of *character-pvar* are not digits of the specified radix, **digit-char-p!!** returns **nil**.

Notice that digit character pvars are always also graphic character pvars.

The argument *character-pvar* must be a character pvar, a string-char pvar, or a general pvar containing only character or string-char elements.

The argument *radix-pvar* must be a positive integer pvar and defaults to **(!! 10)**.

alphanumericp!! *character-pvar* [*Function*]

This function tests *character-pvar* for alphanumeric elements.

In those processors where *character-pvar* is either a digit (of radix 10) or an alphabetic character, **alphanumericp!!** returns **t**. It returns **nil** where this test fails.

The argument *character-pvar* must be a character pvar, a string-char pvar, or a general pvar containing only character or string-char elements.

```
(alphanumericp!! character-pvar)
<=>
(or!! (alpha-char-p!! character-pvar)
      (not!! (null!! (digit-char-p!! character-pvar))))
```

char=!! *character-pvar* **&rest** *more-character-pvars* [*Function*]
char/=!! *character-pvar* **&rest** *more-character-pvars* [*Function*]
char<!! *character-pvar* **&rest** *more-character-pvars* [*Function*]
char>!! *character-pvar* **&rest** *more-character-pvars* [*Function*]
char<=!! *character-pvar* **&rest** *more-character-pvars* [*Function*]
char>=!! *character-pvar* **&rest** *more-character-pvars* [*Function*]

These functions compare the character element of *character-pvar* in each processor against the character elements of each &rest argument pvar in the same processor.

A boolean pvar is returned. It contains **t** in those processors where the test is true and **nil** in those processors where the test is false.

The argument *character-pvar* and each optional &rest argument must be a character pvar, a string-char pvar, or a general pvar containing only character or string-char elements.

Examples:

```
(char<!! (!! #\A) (!! #\B) (!! #\Z) ) => t!!
(char>!! (!! #\z) (!! #\j) (!! #\a) ) => t!!
(char<=!! (!! #\5) (!! #\1) (!! #\5) ) => nil!!
```

The ordering of alphanumeric character pvars used by *Lisp is the ASCII ordering:

```
...0<1<2...<8<9...<A<B<C...<X<Y<Z..<a<b<c...<x<y<z...
```

This ordering is the same as the total ordering produced by applying the function **char-int!!** to such pvars. Notice that this ordering might not be the same as that used by the front-end machine for the scalar equivalents of these character comparison functions. This implementation dependency should be taken into account when character comparisons on front end scalar characters are mixed with the parallel character comparisons described here.

For the purpose of these functions, if any two characters differ in any attribute, they are considered different. Thus,

```
(char=!! (make-char!! (!! #\Q) (!! 0) (!! 0))
         (make-char!! (!! #\Q) (!! 3) (!! 0))) => nil!!
```

This strictness with respect to attributes is relaxed in the following set of functions.

char-equal!! *character-pvar* **&rest** *more-character-pvars* [*Function*]
char-not-equal!! *character-pvar* **&rest** *more-character-pvars* [*Function*]
char-lessp!! *character-pvar* **&rest** *more-character-pvars* [*Function*]
char-greaterp!! *character-pvar* **&rest** *more-character-pvars* [*Function*]
char-not-greaterp!! *character-pvar* **&rest** *more-character-pvars* [*Function*]
char-not-lessp!! *character-pvar* **&rest** *more-character-pvars* [*Function*]

These functions make case-insensitive comparisons between the character element of *character-pvar* in each processor and the character elements of each &rest argument pvar in the same processor. Differences in case, bit, and font attributes are ignored.

A boolean pvar is returned. It contains **t** in those processors where the test is true and **nil** in those processors where the test is false.

The argument *character-pvar* and each optional &rest argument must be a character pvar, a string-char pvar, or a general pvar containing only character and string-char elements.

2.4 Character Control Bit Functions

char-bit!! *character-pvar bit-name-pvar* [*Function*]

This function tests the *bit-name-pvar* bit setting of *character-pvar*.

In those processors where *character-pvar* contains a character element that has the *bit-name-pvar* bit set, **char-bit!!** returns **t**. It returns **nil** where *character-pvar* contains a character element that does not have the *bit-name-pvar* bit set.

The argument *character-pvar* must be a character pvar, a string-char pvar, or a general pvar containing only character and string-char elements.

Unlike its Common Lisp analogue, the argument *bit-name-pvar* must be an integer pvar (either an unsigned-byte or a signed-byte pvar). The following correspondence holds between legal values for the *bit-name-pvar* argument and the recommended Common Lisp control-bit constants:

Common Lisp	*Lisp
:control	(!! 0)
:meta	(!! 1)
:super	(!! 2)
:hyper	(!! 3)

For example:

```
(char-bit!! (!! #\control-x) (!! 0)) => t!!
```

set-char-bit!! *character-pvar bit-name-pvar newvalue-pvar* [*Function*]

This function constructs a copy of *character-pvar* with the *bit-name-pvar* bit set to *newvalue-pvar* in each processor. It returns a pvar containing characters that resemble those in *character-pvar* except that the *bit-name-pvar* bit is set on or off depending on the value of the boolean pvar, *newvalue-pvar.*

The argument *character-pvar* may be a character pvar, a string-char pvar, or a general pvar containing only character or string-char elements.

The argument *bit-name-pvar* must be an integer pvar in the range **(!! 0)** through **(!! 3)**, inclusive. The same correspondence holds between legal values for the *bit-name-pvar* argument to set-char-bit!! and the Common Lisp **control-bit** constants as detailed above for **char-bit!!**.

For example:

```
(set-char-bit!! (!! #\x) (!! 0) t!!) => (!! #\control-x)
(set-char-bit!! (!! #\control-x) (!! 0) t!!) => (!! #\control-x)
(set-char-bit!! (!! #\control-x) (!! 0) nil!!) => (!! #\x)
```

Chapter 3

Array Pvars

*Lisp defines array pvars as the parallel equivalent of Common Lisp arrays with the exception that more stringent restrictions on type and size apply to array pvars than to Common Lisp arrays. *Lisp pvar arrays are pvars containing one array per processor. As with Common Lisp arrays, *Lisp array pvars are stored in row major order.

Whereas Common Lisp includes both general and specialized arrays, *Lisp supports only specialized array pvars. Each element of an array pvar must be a pvar of a given restricted type; array pvars may not contain general pvars. Also, general pvars may not contain array pvars as elements.

Adjustable array pvars are currently not implemented in *Lisp. This means that it is not possible to dynamically alter the dimensions of an array pvar.

Array pvars of variably sized elements are not currently implemented in *Lisp. This means it is an error to attempt to create array pvars containing elements of non-uniform size.

The allowable types for array pvar elements are currently restricted to valid Common Lisp types of fixed size. Thus, array pvar elements may not be declared mutable.

3.1 Array Pvar Type Definition

(pvar (array *element-type dimension-list*)) [*Type*]

This form defines the array pvar type. The *element-type* specifier may be any valid pvar type of fixed size, including the array pvar type itself. The *dimension-list* specifier must be a list of one or more non-negative integers.

Example:

```
(*let (a1)
    (declare (type (pvar (array (unsigned-byte 32) (10 10))) a1))
    a1
)

(*let (a2)
    (declare (type (pvar (array (array boolean (2 2)) (10))) a2))
    a2
)
```

Notice how easy it would be to forget to enclose **a2**'s outer dimension specifier, 10, within a list.

IMPORTANT

The dimension specification within an array type declaration *must* be a list. Be careful not to omit the parentheses when declaring a one-dimensional array.

3.2 Array Pvar Limits

Three *Lisp constants constrain the allowable dimensions of *Lisp array pvars.

***array-rank-limit** [*Constant*]

This is the upper exclusive bound on the number of dimensions a pvar array can have. The number of dimensions specified for a *Lisp array pvar must be less than ***array-rank-limit**. ***array-rank-limit** is guaranteed to be greater than or equal to 8.

Unlike its Common Lisp counterpart, **make-array!!** does not support the following keyword parameters: **:initial-contents**, **:adjustable**, **:fill-pointer**, **:displaced-to**, and **:displaced-index-offset**.

Example:

```
(setq new-array-pvar
      (make-array!! '(2 2 2) :element-type '(complex single-float)
              :initial-element 5.3)
)

(aref (pref new-array-pvar 0) 0 1 0)
=> #C(5.3 0.0)
```

A pvar consisting of a three-dimensional array containing single-precision complex numbers in each processor is defined and bound to the symbol **new-array-pvar**. The value **(!! 5.3)** is ***set** into **new-array-pvar** so that, in all active processors, each array element is initialized. An arbitrary array reference in processor 0 verifies the presence of an initial pvar array element value.

3.3.2 Using !! to Create Array Pvars

The function !! can be used to create an array pvar allocated on the *Lisp stack.

!! *common-lisp-array* [*Function*]

A call to !! with a Common Lisp array as its argument creates and returns a *Lisp array pvar. The resulting array pvar has a copy of all the elements of *common-lisp-array* in each active processor.

If *common-lisp-array* has a fill pointer, it is ignored; all elements of the array are copied into the CM. If *common-lisp-array* is adjustable, the resulting array pvar will nonetheless be of a fixed size equal to that of *common-lisp-array* at the time !! was invoked. Similarly, if *common-lisp-array* is displaced, the elements of the array it is displaced to will be copied into the array pvar, but the array pvar will not itself be displaced.

The type of any array pvar created with !! is determined by the types of the elements in *common-lisp-array*. If *common-lisp-array* contains elements of various types, the *Lisp rules of type coercion apply. These rules closely follow the rules of Common Lisp and are detailed in chapter 8. For example, if a Common Lisp array containing both

integer and floating-point elements is supplied as an argument to !!, the resulting array pvar has elements of type floating-point.

It is an error to call !! on a Common Lisp array containing elements that cannot, according to the *Lisp rules of type coercion, be coerced into a single, fixed-size type. For example, an array containing both characters and integers is not a legal argument to !!.

Nested arrays of arbitrary depth are legal arguments to !!. For instance, an array of arrays of structures is a permissible argument to !!—if that structure was defined using ***defstruct**. Be aware that calling !! with these kinds of nested arrays can be a very slow operation.

Examples:

```
(*let ((parallel-array (!! #(1 2 3))))
    (declare (type
      (pvar (array (unsigned-byte 8) (3))) parallel-array))
    (do-something-to array-pvar))
```

A one-dimensional Common Lisp array of three elements is duplicated in all active processors and **parallel-array** is bound to the result.

```
(*let ((points (!! #(#(2 4)#(6 12)#(7 16)#(5 20)#(2 56)))))
     (declare (type
       (array-pvar (vector (unsigned-byte 8) 2) (5)) points))
      (do-something-to points))
```

A five-element Common Lisp array of two-element vectors is duplicated in all active processors and **points** is bound to the result.

3.3.3 Using *let and *let* to Create Array Pvars

Array pvars can be allocated on the *Lisp stack by declaring them appropriately from within a ***let** or a ***let*** form. Be careful: when allocating an array using ***let** or ***let***, don't forget to declare the type of the pvar because undeclared pvars that have held any other type of data cannot hold arrays.

Examples:

```
(*let (foo)
  (declare (type (pvar (array single-float (3 3))) foo))
  (*setf (aref!! foo (!! 0) (!! 1)) (!! 2.3))
  (aref (pref foo 0) 0 1)
  )

=> 2.3


(*let ((bar (make-array!! '(3 3 3) :element-type '(pvar boolean)
                                   :initial-element t)
       ))
  (declare (type (pvar (array boolean (3 3 3))) bar))
  (ppp bar :end 1)
  )

=>
#3A(((T T T)(T T T)(T T T))((T T T)(T T T)(T T T))((T T T)(T T T)
(T T T)))
```

3.3.4 Using allocate!! to Create Array Pvars

Array pvars may be allocated on the *Lisp heap by using **allocate!!**.

Example:

```
(setq baruch (allocate!! (!! #(1 2 3)) nil
               '(pvar (array (unsigned-byte 8) (3)))))

(ppp baruch :end 2)


=> #(1 2 3) #(1 2 3)
```

3.3.5 Using *defvar and *proclaim to Create Array Pvars

Array pvars may be allocated on the *Lisp heap by using ***defvar** and ***proclaim**. Be careful: when allocating an array pvar using ***defvar**, be sure to first declare the type of pvar using ***proclaim**. Undeclared pvars cannot hold arrays.

Examples:

```
(*proclaim '(type (pvar (array character (3 4 5))) fum))

(*defvar fum (make-array!! '(3 4 5)
                           :element-type '(pvar character)
                           :initial-element #\L))

(ppp (aref!! fum (!! 1) (!! 2 ) (!! 0 )) :end 10)

=> #\L #\L #\L #\L #\L #\L #\L #\L #\L #\L


(*proclaim '(type (pvar (array (unsigned-byte 8) (3))) fee))
(*defvar fee)
(*set fee (!! #(1 2 3)))
(ppp fee :end 3)

=> #(1 2 3) #(1 2 3) #(1 2 3)
```

3.3.6 Array Pvars with Dynamically-Determined Dimensions

It is possible to allocate array pvars whose dimensions are known only at run time. A properly constructed array pvar type declaration within a ***let** or a ***let*** form is used. The dimensions specification of the declaration may be given in one of two ways:

1. A list of dimension values, $(x\,y\,z)$, may be given, such that x, y, and z each evaluate to integers at run time.

2. A variable may be named. Its value at run time must be a list of integers.

Examples:

```
(defun make-2d-array-pvar (x y)
  (*let (temp-array)
     (declare (type (pvar (array single-float (x y))) temp-array))
   temp-array))


(*proclaim '(type (pvar (array single-float (5 5))) 5-by-5))
(*defvar 5-by-5)
(*set 5-by-5 (make-2d-array-pvar 5 5))
```

The formal parameters `x` and `y` are bound to specific values upon invocation of **make-2d-array**. The dimensions of **temp-array** are then determined upon execution of the form.

Example:

```
(defun good-make-array-pvar (input-scalar-array)
    (let ((dims (array-dimensions input-scalar-array)))
      (*let (temp)
        (declare (type (pvar (array single-float dims)) temp))
     temp)))


(defun bad-make-array-pvar (input-scalar-array)
    (*let (temp)
      (declare (type (pvar (array single-float
                (array-dimensions input-scalar-array)))
       temp))
    temp))
```

Any array pvar declaration form expects a list of integers specifying array dimensions.

The **bad-make-array-pvar** function definition is in error because it places the form **(array-dimensions input-scalar-array)** inside the declare form. The declaration should instead contain a list of integer dimensions or a symbol bound to such a list.

The **good-make-array-pvar** function definition works properly because the symbol **dims** is bound to a list of integers: the result of **(array-dimensions input-scalar-array)**. The symbol **dims** is then supplied to the **declare** form, which, when executed, finds **dims** properly bound to a list of integers.

3.4 Creating Vector Pvars

Just as Common Lisp vectors are equivalent to one-dimensional Common Lisp arrays, *Lisp vector pvars are equivalent to one-dimensional array pvars. Unlike Common Lisp, which provides both typed and general vectors, *Lisp does not support vector pvars that have elements of type t. *Lisp supports only typed vectors.

(pvar (vector *element-type length*)) [*Pvar Type*]
(vector-pvar *element-type length*) [*Pvar Type*]

These two forms may be used interchangeably in typed vector pvar declarations. The *element-type* must be a Common Lisp scalar type. The *length* defines the number of *element-type* elements contained in each active processor.

typed-vector!! *component-type* **&rest** *pvars* [*Function*]

The function **typed-vector!!** creates and returns a one-dimensional array pvar of type *component-type*. Initial contents are copied from the supplied *pvars*. The *n*th pvar argument is ***set** into the *n*th vector element.

The *component-type* argument describes the pvar type of the vector pvar's components—not the type of the component vectors' elements.

Notice that,

```
(typed-vector!! '(pvar single-float) (!! 1.0) (!! 2.0) (!! 3.0))

<=>

(*let (temp)
    (declare (type (pvar (array single-float (3))) temp))
    (dotimes (j 3)
   (*setf (aref!! temp (!! j))(!! float (1+ j)))
    ))
)
```

That is, a call to **typed-vector!!** is equivalent to a ***let** form that declares and then initializes a one-dimensional array pvar.

Chapter 10 of this supplement describes experimental *Lisp operations that manipulate numeric vectors.

3.5 Operations Returning Array Pvar Information

***array-element-type** *array-pvar* [*Function*]

This function returns a scalar type specifier for the elements *array-pvar*. If no processors are active, ***array-element-type** nonetheless returns the proper element type.

***array-rank** *array-pvar* [**Defun*]

This operation returns an unsigned integer equal to the number of dimensions in *array-pvar*.

array-rank!! *array-pvar* [*Function*]

This function returns a pvar containing, in each processor, an unsigned integer equal to the number of dimensions in *array-pvar*.

***array-dimension** *array-pvar axis-scalar* [**Defun*]

This operation returns an unsigned integer equal to the size of the *array-pvar* dimension referenced by *axis-scalar*.

The argument *axis-scalar* must be an unsigned integer less than the rank of *array-pvar*.

array-dimension!! *array-pvar axis-scalar-pvar* [*Function*]

This function returns a pvar containing, in each processor, an unsigned integer equal to the size of the a*xis-scalar-pvar* dimension of *array-pvar*.

The argument *axis-scalar-pvar* must be a pvar containing, in each processor, an unsigned integer less than the rank of *array-pvar*.

***array-dimensions** *array-pvar* [**Defun*]

This operation returns a list enumerating the dimensions of *array-pvar*. This list is of length **(*array-rank array-pvar)**.

array-dimensions!! *array-pvar* [*Function*]

This function returns a vector pvar containing, in each processor, a vector such that the value of the *n*th element of the vector is the extent of the *n*th dimension of *array-pvar* in that processor.

***array-total-size** *array-pvar* [**Defun*]

This operation returns an unsigned integer equal to the total number of *array-pvar* elements contained in each processor.

Notice that the result is not the total number of array elements in all processors. Rather, it is the number of elements in a single processor and this count is the same for all processors.

array-total-size!! *array-pvar* [*Function*]

This function returns, in each processor, an unsigned integer equal to the total number of *array-pvar* elements contained in that processor.

array-in-bounds-p!! *array-pvar* **&rest** *pvar-subscripts* [*Function*]

This function returns a boolean pvar with **t** in every processor where *pvar-subscripts* represents a valid reference to *array-pvar* and **nil** elsewhere.

array-row-major-index!! *array-pvar* **&rest** *pvar-subscripts* [*Function*]

This function returns an unsigned pvar identifying the row-major index represented by *pvar-subscripts* in each processor.

The *pvar-subscripts* arguments must be valid *array-pvar* subscripts. Each of these &rest arguments corresponds to a dimension of *array-pvar*; they must be given in order, starting with dimension 0. The number of *pvar-subscripts* arguments must equal the rank of *array-pvar*.

3.6 Accessing Array Elements

aref!! *array-pvar* **&rest** *subscript-pvars* [*Function*]

This function returns a pvar on the *Lisp stack. The result pvar contains, in each processor, a copy of the *array-pvar* element specified by *subscript-pvars*. The type of the returned pvar is the same as the element type of *array-pvar*.

The argument *array-pvar* must be a *Lisp array pvar.

One *subscript-pvar* argument must be given for each dimension of *array-pvar*. Each *subscript-pvar* must contain non-negative integers within the range of indices for that dimension. The number of arguments given as *subscript-pvars* must equal the rank of *array-pvar*.

Examples:

```
(aref!! 2by5-array-pvar (!! 1) (!! 4))
```

This returns a pvar containing, in each processor, a copy of the element (1,4) of *2by5-array-pvar* found in that processor.

The function **aref!!** may be used in conjunction with ***setf** to selectively set the value of individual array pvar elements. For example,

```
(*setf (aref!! 2by5-array-pvar (!! 1) (!! 4)) (!! 2))
```

sets element **(1,4)** of **2by5-array-pvar** in each processor to 2.

```
(*let (foo)
  (declare (type (pvar (array single-float (3 3))) foo))
  (*setf (aref!! foo (!! 0) (!! 1)) (!! 2.3))
  foo)
```

This form declares **foo** to be a two-dimensional single-float array pvar, sets the first element of the second row in each processor to 2.3, and return **foo**.

IMPORTANT

To modify array pvar elements, use (***setf (aref!! ...)**). This is the only way to modify an individual array pvar element. It is an error to use the construct (***set (aref!!...)**. Using this erroneous construct only results in modifying a *copy* of the array pvar element. The original array pvar would not be changed.

3.6.1 Indirect Addressing of Array Pvar Elements

In *Lisp, the term *indirect addressing* is used to refer to pvar array referencing that uses different index values in different processors.

If the subscript arguments to **aref!!** are textually of the form **(!! integer)**, **aref!!** extracts the values of array elements from uniform coordinates in all processors. If the subscript arguments to **aref!!** contain different values in different processors, indirect addressing is said to take place. Indirect addressing references array elements indirectly by deriving element positions (addresses) from pvars that hold various values in different processors.

As an illustration, suppose three pvars exist, **X**, **Y**, and **C**:

```
(ppp X :end 4) => 0 1 1 0

(ppp Y :end 4) => 0 4 1 2

(ppp C :end 4) => #\A #\B #\C #\D

(*let (letters)
    (declare (type (pvar (array character (2 5))) letters))
    (*setf (aref!! letters X Y) C)
)
```

In processor 0, the **letters** array element **(0,0)** is set to **A**. In processor 1, the **letters** array element **(1,4)** is set to **B**, and so on.

3.6.2 Accessing Array Pvar Elements Directly: Aliasing

The result of calling **aref!!** is always a copy of the contents of an array element pvar and is always allocated on the *Lisp stack. To create a pvar referring to the same bits in Connection Machine memory as the bits of an array element pvar, use the macro **alias!!** in conjunction with **aref!!**.

alias!! *array-reference* [*Macro*]

This operation accesses the pvar object referenced by *array-reference*. The **alias!!** operation should be used when passing a pvar array element to a function that alters the array.

The *array-reference* argument to **alias!!** should be an **aref!!** call; it may not be an array reference that uses indirect addressing. An error is signaled if an attempt is made to use indirect addressing within an **alias!!** form.

Examples:

```
(defun modify-foo-element (pvar value) (*set pvar value))

(defun in-error ()
  (*let (foo)
    (declare (type (pvar (array (unsigned-byte 8) (3))) foo))
    (modify-foo-element (aref!! foo (!! 0)) (!! 3))
    ))

(defun correct ()
  (*let (foo)
    (declare (type (pvar (array (unsigned-byte 8) (3))) foo))
    (modify-foo-element (alias!! (aref!! foo (!! 0))) (!! 3))
    ))
```

The **in-error** function is in error because it tries to ***set** a temporary pvar on the *Lisp stack. The form **(aref!! foo (!! 0))** returns a pvar allocated on the *Lisp stack and containing a copy of the data from the 0th element of **foo** in each processor. The function **modify-foo-element** then tries to ***set** this temporary pvar. If this were allowed, the intended result would not be obtained: ***set** would change the data on the *Lisp stack, not the data in the **foo** array pvar.

The **correct** function works because the **alias!!** macro returns a pvar which can be modified: it points to the Connection Machine memory locations that contain element (0,3) of array pvar **foo** in each processor.

3.6.3 Sideways Arrays: an Experimental Feature

Indirect addressing of array pvar elements is slower than array referencing that uses index pvars containing the same values in each processor. To speed up indirect addressing, a new, experimental *Lisp function named ***sideways-array** is provided.

***sideways-array** *array-pvar* [*Function*]

The function ***sideways-array** forces *array-pvar* to be addressed in a *sideways* ordering. Calling ***sideways-array** on an array that is already sideways returns it to a processorwise ordering. This function is executed for side effect; no useful value is returned.

Turning an array sideways allows special Connection Machine hardware to read array pvar elements that are not uniformly positioned across processors. Indirect addressing works significantly more quickly on sideways arrays than on normal arrays.

The *array-pvar* argument must be an array pvar that contains elements whose lengths are powers of 2. This restriction may be lifted in the future. The ***sideways-array** function is most efficient when using array elements that are 32 bits long.

Elements of a sideways array may be accessed with **sideways-aref!!**. Data may be stored into a sideways array using **(*setf (sideways-aref!!...))**.

The following restrictions apply to sideways arrays.

- A sideways array may not be ***set**.
- A sideways array may not be used as the *pvar-expression* source argument to **pref!!**.
- A sideways array may be used as neither the *value-pvar* source nor as the *dest-pvar* destination argument to ***pset**.
- A sideways array may not be read out to the front end.

  Before performing any of the above operations on a sideways array, the array must be returned to its normal state by executing a second call to ***sideways-array**.
- Using ***sideways-array** on an array that is defined as a slot of a ***defstruct** is not supported.

sideways-aref!! *array-pvar* **&rest** *subscript-pvars* [*Function*]

This function works just like **aref!!**, but it is a special accessor defined to operate on sideways arrays only. Requiring this distinction allows the *Lisp compiler to generate efficient code to reference sideways arrays without requiring declarations that identify arrays as sideways.

The argument *array-pvar* must be a sideways array.

One *subscript-pvar* argument must be given for each dimension of *array-pvar*. Each *subscript-pvar* must contain non-negative integers within the range of indices for that dimension. The number of arguments given as *subscript-pvars* must equal the rank of *array-pvar*. Unless one or more *subscript-pvar* arguments contain non-uniform values across processors, there is no benefit to using this function.

To obtain maximum performance when addressing an array indirectly, turn the array sideways by using the function ***sideways-array**. Then, instead of using **aref!!** to read from that array and **(*setf (aref!!...))**to write to it, use **(sideways-aref!! ....)** and **(*setf (sideways-aref!! ...))**.

3.7 Logical Operations on Bit Array Pvars

Parallel equivalents of the Common Lisp bit-wise logical operations are provided in *Lisp to operate on bit-array pvars. An array pvar is considered a *bit-array* pvar if and only if its element type is **(pvar (unsigned-byte 1))**.

bit-and!! *bit-array-pvar-1 bit-array-pvar-2* [*Function*]
&optional *bit-array-result-pvar*

bit-ior!! *bit-array-pvar-1 bit-array-pvar-2* [*Function*]
&optional *bit-array-result-pvar*

bit-xor!! *bit-array-pvar-1 bit-array-pvar-2* [*Function*]
&optional *bit-array-result-pvar*

bit-eqv!! *bit-array-pvar-1 bit-array-pvar-2* [*Function*]
&optional *bit-array-result-pvar*

bit-nand!! *bit-array-pvar-1 bit-array-pvar-2* [*Function*]
&optional *bit-array-result-pvar*

bit-nor!! *bit-array-pvar-1 bit-array-pvar-2* [*Function*]
&optional *bit-array-result-pvar*

bit-andc1!! *bit-array-pvar-1 bit-array-pvar-2* [*Function*]
&optional *bit-array-result-pvar*

bit-andc2!! *bit-array-pvar-1 bit-array-pvar-2* [*Function*]
&optional *bit-array-result-pvar*

bit-orc1!! *bit-array-pvar-1 bit-array-pvar-2* [*Function*]
&optional *bit-array-result-pvar*

bit-orc2!! *bit-array-pvar-1 bit-array-pvar-2* [*Function*]
&optional *bit-array-result-pvar*

A helpful chart detailing the meaning and effect of each of these functions may be found in chapter 17 of *Common Lisp: The Language*.

Each of these functions perform a logical bit-wise operation on the contents of the first two arguments. The result is a bit-array pvar of the same rank and dimensions as *bit-array-pvar-1* and *bit-array-pvar-2*.

It is an error if both required arguments are not bit-array pvars of identical rank and dimensionality.

If supplied, the optional argument may be given as **t**, as **nil**, or as a bit-array pvar with the same rank and dimensions as the required arguments. It defaults to **nil**. If **nil** or no value is supplied for the optional argument, the operation returns a bit-array pvar on the *Lisp stack. If a bit-array pvar is supplied as the value of the optional argument, the result of the operation is destructively stored in it. If **t** is supplied as the value of the optional argument, results are destructively stored in the first argument, *bit-array-pvar-1*.

bit-not!! *bit-array-pvar* [*Function*]
&optional *bit-array-result-pvar*

This function inverts all the bits in *bit-array-pvar*. The result is a bit-array pvar of the same rank and dimensions as *bit-array-pvar*.

The optional argument *bit-result-array-pvar* may be used to specify where the result of **bit-not!!** should be placed. If supplied, *bit-result-array-pvar* may be given as **t**, as **nil**, or as a bit-array pvar with the same rank and dimensions as *bit-array-pvar*. It defaults to **nil**, indicating that **bit-not!!** returns a bit-array pvar on the *Lisp stack. If a bit-array pvar is supplied as the value of *bit-result-array-pvar*, the **bit-not!!** result is destructively stored in it. If **t** is supplied as the value of *bit-result-array-pvar*, results are destructively stored in the required argument, *bit-array-pvar*.

3.8 Mapping Functions Over Array Pvars

***map** *function* **&rest** *array-pvars* [*Function*]

***map** applies *function* repeatedly to a list composed of array element pvars, with one element from each *array-pvar* supplied. The supplied function, *function*, is applied as many times as there are elements in the smallest of the supplied *array-pvars*. Each **&rest** argument is processed in row-major order. Thus, the *n*th call to *function* gets passed an alias to the *n*th element of each *array-pvar*, where *n* is taken to be the row-major ordering.

***map** returns **nil**; it is executed for side effect.

Example:

Suppose we have two matrices and we wish to add the two matrices together element by element, multiplying the result of the addition by a constant, and storing the overall result back in the first matrix. The following code illustrates this:

```
(*proclaim '(type (pvar (array single-float (3 3)))
           matrix1 matrix2))
(*defvar matrix1)
(*defvar matrix2)

(defun *map-example (single-float-constant)
  (*locally
    (declare (type single-float single-float-constant))
```

```
(*map
  #'(lambda (element1 element2)
 (*locally
   (declare (type single-float-pvar element1 element2))
   (*set element1 (*!! (+!! element1 element2)
                       (!! single-float-constant)))
   ))
  matrix1
  matrix2
  )))
```

3.9 Notes on Using Array Pvars

(1) The **pref** operation works on array pvars.

The constructs **pref** and **(*setf (pref ...))** work on array-pvars in the expected manner. For example,

```
(*setf (pref (aref!! a-vector-pvar (!! 0)) 0) 5)
```

sets the 0th element of **a-vector-pvar** in the 0th processor to 5.

(2) Nested arrays and structures are allowed.

It is possible to nest array and structure references to any level by using ***setf**. Thus,

```
(*setf (pref (aref!! (aref!! (structure-slot-A!! x) (!! 2))
        (!! 3) (!! 4))
       10) 5)
```

stores 5 in slot **A** of the structure **x** found in the second element of the vector stored in the array element indexed by **(3,4)** in processor 10. (Information about structure pvars may be found in chapter 4.)

(3) Remember !! makes copies.

It is true that

```
(*all (equalp common-lisp-array (pref (!! common-lisp-array) 0)))
```

but it is never true that

```
(*all (eq common-lisp-array (pref (!! common-lisp-array) 0)))
```

In other words, putting a front-end array into Connection Machine processors using !! and then reading out an instance of it using **pref**, will not result in the original array (**eq**) but in a copy (**equal**) of that array.

(4) The function **equalp!!** can be used on array pvars to test for element-by-element equivalence. See section 7.1 for a complete definition of **equalp!!**.

(5) ***map** gets around the restrictions on **scan!!** for array pvars.

The only scan operation allowed to be performed on array pvars is **copy!!**. In order to apply other scan operators to array pvars, use ***map** in conjunction with **scan!!** as illustrated below.

```
(*map
  #'(lambda (dest source) (*set dest (scan!! source '+!!)))
  array2
  array1
  )
```

This performs an element-wise plus scan on **array1** and puts the results, element-wise, into **array2**.

Chapter 4

Structure Pvars

*Lisp implements parallel equivalents of the Common Lisp structure definition capabilities described in chapter 19 of *Common Lisp: The Language*. While Common Lisp provides the function **defstruct** to create user-defined data structures, *Lisp provides ***defstruct** to create user-defined structure pvars. There are a few differences between *Lisp and Common Lisp with respect to structures. These are noted in this chapter.

As with Common Lisp structures, a structure pvar definition creates a new, named, aggregate data type. Constructor, accessor, and assignment operations are automatically defined when a structure pvar data type is defined.

*Lisp structure pvars are consistent with Common Lisp structures with respect to nesting and layering. Structure pvars may be nested to any depth. That is, one structure pvar may contain other structures pvars, which may contain other structure pvars, and so forth. Structure pvar definitions may also be layered: one structure definition may include all of another structure definition. Layered definitions are restricted to one inclusion per structure pvar. Nonetheless, there is no limit on the depth of a structure type hierarchy created with layered inclusions.

Like a Common Lisp structure definition, a *Lisp structure pvar definition automatically defines slot accessor functions. Unlike Common Lisp structures, structure pvar accessor functions do not return pointers to structure pvar contents. Instead, a *copy* of the structure pvar slot contents is always returned by an accessor function. The *Lisp function **alias!!** is provided to access the Connection Machine data bits representing structure pvar contents.

*Lisp does not support ***defstruct** options analogous to the Common Lisp **defstruct** options **:type**, **:named**, and **:initial-offset**.

*Lisp extends the slot options by adding the Connection Machine-specific ***defstruct** slot options **:cm-initial-value** and **:cm-uninitialized-p**.

4.1 Defining Structure Pvars

The macro ***defstruct** defines structure pvars types in *Lisp. Using a ***defstruct** form has the interesting effect of defining both a Common Lisp scalar structure data type and a Connection Machine parallel structure data type. Further, ***defstruct** defines both scalar and parallel constructor, accessor, and assignment operations for the new data types it creates. Thus, whereas most *Lisp operations behave like parallel versions of Common Lisp operations, ***defstruct** performs the Common Lisp **defstruct** operation as well as a parallel version of that operation.

The double functionality of ***defstruct** allows structures to be passed back and forth between the Connection Machine system and the front-end computer. Once a structure pvar has been defined with ***defstruct**, a structure pvar of that type may be allocated on the Connection Machine. Given a structure pvar, an individual structure may be extracted from a single processor using **pref!!** and copied to the front end. Similarly, an instance of the structure may be created on the front end and broadcast to the Connection Machine processors.

4.1.1 What *defstruct Does

A ***defstruct** form does the following:

- defines a new pvar type, which may be used in pvar type declarations
- defines a parallel constructor function
- defines pvar accessors to access the slots of the structure pvar
- defines ***setf** methods to set the slots of the structure pvar
- defines a *Lisp predicate to test whether or not a pvar contains structures of the newly defined type
- defines a front-end **defstruct** object corresponding to the ***defstruct**
- allows **!!**, ***setf** of **pref**, **array-to-pvar**, **pvar-to-array**, **array-to-pvar-grid**, and **pvar-to-array-grid** to take a front-end **defstruct** object as the value stored in a structure pvar of the corresponding type

Consider the following example.

```
(*defstruct asteroid
  (diameter 1 :type (unsigned-byte 16))
  (mappedp nil :type boolean)
  )
```

This automatically defines the following **defstruct**.

```
(defstruct asteroid
  (diameter 1 :type (unsigned-byte 16))
  (mappedp nil :type boolean)
  )
```

The data types **asteroid** and **(pvar asteroid)** are defined by these forms. (Note: *Lisp defines boolean as equivalent to **(member t nil)**.)

The following functions are automatically defined by the above ***defstruct** form:

Parallel	**Scalar**
make-asteroid!! &key **(:diameter (!! 1)** **(:mappedp nil!!)**	**make-asteroid &key** **(:diameter 1)** **(:mappedp nil)**
asteroid-diameter!! *asteroid-pvar*	**asteroid-diameter** *asteroid*
asteroid-mappedp!! *asteroid-pvar*	**asteroid-mappedp** *asteroid*
asteroid-p!! *pvar*	**asteroid-p** *symbol*
	copy-asteroid *some-asteroid*

The scalar functions are the familiar Common Lisp constructor, accessor, and predicate functions produced by defstruct. The parallel versions are described below.

The constructor function **make-asteroid!!** makes an asteroid structure pvar of type **(pvar asteroid)**. Calling this function creates, in each processor, an asteroid instance composed of slots **diameter** and **mappedp**. The slot pvars **diameter** and **mappedp** are initialized to the default values **(!! 1)** and **nil!!**, unless alternative values are supplied to the keywords **:diameter** and **:mappedp**. For example,

```
(*proclaim '(type (pvar asteroid) cm-wally))
(*defvar cm-wally (make-asteroid!!))
```

creates an asteroid structure pvar named **cm-wally**. The new asteroid, **cm-wally**, consists of an asteroid structure in all processor. An equivalent method of making **cm-wally** is:

```
(*proclaim '(type (pvar asteroid) cm-wally))
(*defvar cm-wally (!! (make-asteroid)))
```

The accessor function **asteroid-diameter!!** returns a pvar of type **(pvar (unsigned-byte 16))**. Similarly, **asteroid-mappedp!!** returns a pvar of type **(pvar boolean)**. These return values are copies of the slot pvars they access and are allocated on the *Lisp stack.

```
(*let (a-random-asteroid)
  (declare (type (pvar asteroid) a-random-asteroid))
  (*setf (asteroid-diameter!! a-random-asteroid) (!! 7))
  a-random-asteroid)
```

To set the value of a structure pvar slot in a particular processor, the function ***setf** is composed with **pref**. Be careful: when allocating a structure using ***let** or ***let***, don't forget to declare the type of the pvar, because an undeclared pvar that has held any other type of data cannot hold a structure pvar.

Given an asteroid structure pvar, **cm-wally**,

```
(*setf (pref (asteroid-diameter!! cm-wally) 25) 15)
```

stores the integer 15 in **cm-wally**'s slot, **diameter**, in processor 25 only.

It is also possible to create a scalar asteroid.

```
(setq wally (make-asteroid :diameter 66 :mappedp t))
```

Given **wally**, ***setf** can be used to make the **cm-wally** asteroid structure contained in one processor be a copy of **wally**:

```
(*setf (pref cm-wally 5) wally)
```

Here are some other examples of code using asteroids:

```
(*proclaim '(type (pvar asteroid) another-asteroid))
(*defvar another-asteroid (make-asteroid!! :diameter (!! 5)))


(setq asteroid-in-heap
  (allocate!! (make-asteroid!! :mappedp (!! t)) nil
   '(pvar asteroid)))


(*when (not!! (zerop!! (self-address!!)))
  (*setf (pref!! cm-wally (1-!! (self-address!!))) cm-wally)
```

When allocating a structure pvar using ***defvar**, be sure to first declare the type of pvar using ***proclaim**. Undeclared pvars that have held any other type of value cannot hold structures.

4.1.2 Formal *defstruct Definition

defstruct** (*struct-name* {*options*}**) {*slot-descriptor*}+ [*Macro*]

This macro defines an aggregate pvar data type as well as an aggregate scalar data type. Components of the structure, called slots, are also defined. A general call to ***defstruct** has the following format.

```
(*defstruct (struct-name option-1 option-2 ... option-n)
            documentation
            slot-description-1
            slot-description-2
            ...
            slot-description-n)
```

The first argument to ***defstruct** is a list composed of *struct-name* followed by a series of associative lists specifying ***defstruct** options.

The argument *struct-name* must be a symbol. It becomes the name of a new data type with both scalar and parallel versions:

(pvar struct-name) [*Pvar Type*]
struct-name [*Scalar Type*]

The new parallel *Lisp type specifier generated by ***defstruct** can be used in declare forms after ***let**, ***let***, and ***defun**, in **the** forms, in ***proclaim** statements, and in **allocate!!** function calls.

If a call to ***defstruct** does not supply any options, the first argument to ***defstruct** is simply the symbol **struct-name** and need not be enclosed in a list.

If supplied, the *option-1 ... option-n* arguments must be chosen from among the keyword-value pairs described below in section 4.6.1, under the heading "Options to *Defstruct." Each option has the form:

(*keyword* **&rest** *values*)

Most of the Common Lisp **defstruct** options have ***defstruct** equivalents.

The *documentation* argument is optional. If supplied, it must be a string.

Any call to ***defstruct** is required to include at least one *slot-description*. Each *slot-description* is of the form:

```
(slot-name  default-init
   slot-option-name-1  slot-option-value-1
   slot-option-name-2  slot-option-value-2
    . . .
    )
```

Here, *slot-name* is a symbol used to identify one component of the structure *struct-name*. It is an error for two slots to have the same name.

The value of *default-init* must be a form that returns a valid scalar value conforming to the type of the slot, as specified by the **:type** slot option.

The *slot-option-names* and *slot-option-values* are keyword-value pairs. (See section 4.6.2 for a complete list of slot options.) The only ***defstruct** slot options of general interest are **:cm-initial-value**, **:cm-uninitialized-p**, and **:type**. Be aware that it is an error to provide both a **:cm-initial-value** form and to specify **:cm-uninitialized-p** as **t**. Also note that the **:type** slot option is mandatory: it must be given one argument, a valid Common Lisp type specifier which, when turned into a pvar specifier by forming **(pvar** *valid-lisp-type*), must be a valid *Lisp type specifier. The **:type** slot option specifier defines the type of the pvar contained in slot *slot-name*.

The **make-struct-name!!** function takes one argument for each slot in the ***defstruct**. Each of these arguments are keyword arguments called by the same name as their slot. The **make-struct-name!!** function will initialize each slot of the created structure in the following manner:

- If the keyword argument is provided, its value is copied into the slot.
- If no keyword argument is provided and there is a **:cm-initial-value** slot option, that slot option is evaluated and the result is copied into the slot.
- If no keyword argument is provided, and no **:cm-initial-value** option is provided, and no **:cm-unitialiazed-p** option is provided, then the slot value defaults to the value of *default-init*. To accomplish this slot pvar initialization, ***defstruct** replicates the value returned by the *default-init* form using the function !!.
- If no keyword argument is provided and the slot option **:cm-uninitialized-p** is provided, then the slot remains uninitialized.

For instance,

```
(*defstruct baz
  (only-slot 0 :type (unsigned-byte 8))
  )

(make-baz!!)
```

creates an instance of a **baz** pvar with its **only-slot** initialized to **(!! 0)**.

Any type specifier given as a **:type** slot option must specify data of fixed, known size. It is an error to specify a slot as a general or as a mutable pvar. Thus,

```
(*defstruct quidly
  (fee (make-array 10 :initial-value 0)
       :type (array (unsigned-byte 16) (10))))
```

contains a valid type specifier of fixed, known size: 160 bits. On the other hand,

```
(*defstruct queezy
  (bad 0 :type (unsigned-byte *)))
```

is in error because it attempts to specify a structure slot of type mutable unsigned-byte, which is by definition of unknown size. It is interesting to note that

```
(*defstruct ok
  (fine 0 :type (unsigned-byte 32)
          :cm-type (pvar (unsigned-byte some-expression))))
```

contains a legal type specifier, provided that **some-expression** evaluates to an integer constant at run time.

The slot option **:cm-initial-value** takes one argument: an expression that returns a pvar.

For instance,

```
(*defstruct xyz
  (a 0 :type (unsigned-byte 32)
       :cm-initial-value (self-address!!))
  )
```

```
(*proclaim '(type (pvar (structure xyz)) cm-xyz))
(*defvar cm-xyz (make-xyz!!))
(setq fe-xyz (make-xyz))
```

initializes slot **a** of **cm-xyz** in each processor to its processor number and initializes slot **a** of **fe-xyz** to 0.

The slot option **:cm-uninitialized-p** takes one argument, either **t** or **nil** and defaults to **nil**. If the value is **t**, then, when pvars of type *struct-name* are created using **make-***struct-name*, the slot is not initialized. This means the slot value is indeterminate and it is an error to access the value without first setting it. Using this option for some or all slots in a ***defstruct** call makes allocation of any structure pvars of the given type faster.

For example,

```
(*defstruct fos
  (only-slot 0 :type (unsigned-byte 32) :cm-uninitialized-p t))

(*let ((no-vals (make-fos!!))
       (vals (make-fos!! :only-slot (!! 5))))
  (declare (type (pvar fos) no-vals vals))
...)
```

defines **no-vals** as a structure pvar of type **fos** that does *not* store **(!! 0)** in its **only-slot** pvar and **vals** as a **fos** structure pvar that stores **(!! 5)** in its only-slot pvar.

4.2 Structure Inheritance

The ***defstruct** option generally considered most interesting is the **:include** option. This option allows a structure definition to subsume one other structure definition.

The ***defstruct :include** option takes one argument, a symbol, which must be the name of a structure pvar definition created by a previous call to ***defstruct**. The specified structure pvar definition is included at the beginning of the structure pvar being defined by the current ***defstruct**. The resulting structure pvar definition behaves as though all of the slots of the included structure pvar definition were specified textually before the slots of the new structure pvar. The accessors of the included structure pvar correctly access the slots of the new structure pvar.

NOTE: At most one **:include** option can be provided per ***defstruct** definition.

For example:

```
(*defstruct auto
  (doors 4 :type (unsigned-byte 3))
  (color #\R :type character))


(*defstruct (sports-car (:include auto))
  (number-of-speeding-tickets 6 :type (unsigned-byte 4)))
```

Defines the *Lisp parallel structure accessors **auto-doors!!**, **auto-color!!**, and **sports-car-doors!!**, **sports-car-color!!**, and **sports-car-number-of-speeding-tickets!!**. The accessors **auto-doors!!** and **sports-car-doors!!** perform identically on a structure pvar of type **sports-car**. However, a program is in error if it calls the function **sports-car-doors!!** on a structure pvar of type **auto**. This is intuitively true: all **sports-car**s are **auto**s but not all **auto**s are **sports-car**s. In other words, a child-type structure satisfies the predicate of its parent-type and can be accessed with its parent's accessors. However, a parent-type structure does not satisfy the predicates of its child-types, nor can it be accessed with its children's accessors.

4.3 Referencing and Modifying Structure Pvars

The constructs **pref** and ***setf** work on structure pvars and may be composed as: **(*setf (pref ...))**.

Here is an example using nested parallel structures. Consider a parallel structure, **foo**, which has a slot, **a**. Slot **a** is a parallel structure, **bar**. If **bar** has a slot, **b**, and **b** is a one-dimensional array, then

```
(*setf (pref (aref!! (bar-b!! (foo-a!! x)) (!! 0)) 0) 5)
```

sets the array element at index 0 within slot **b** of structure **bar** within slot **a** of structure **foo** in processor 0 to the value 5.

IMPORTANT

To modify structure pvar elements, use ***setf** on the result of a ***defstruct** accessor function. This is the only way to modify an individual structure slot. It is an error to use the construct (***set** (*slot-a!! slot-pvar*)...), where *slot* is the name of a ***defstruct**. Using this erroneous construct only results in modifying a *copy* of the structure slot. The original structure pvar would not be changed.

4.3.1 Accessing Structure Pvar Contents Directly: Aliasing

The result of calling a structure pvar accessor function is always a copy of the contents of the slot accessed. To create a pvar that refers to the same bits in Connection Machine memory as the bits of the slot, the macro **alias!!** must be used. This is useful if it is desirable to pass a structure slot pvar to a function that alters a structure slot pvar.

alias!! *slot-accessor* [*Macro*]

This operation accesses the pvar object referenced by *slot-accessor.*

The *slot-accessor* argument to **alias!!** should be a call to a slot accessor function created by ***defstruct**.

The following code illustrates how to use **alias!!** with structure pvars.

```
(*defstruct patient
  (id-no 0 :type (unsigned-byte 8))
  (doctor 0 :type (unsigned-byte 8))
  (sick-p t : type boolean))

(defun in-error ()
  (*let (ellen)
    (declare (type (pvar patient) ellen))
    (modify-patient-slot (patient-sick-p!! ellen) nil!!)
    ))
```

```
(defun correct ()
  (*let (ellen)
    (declare (type (pvar patient) ellen))
    (modify-patient-slot
      (alias!! (patient-sick-p!! ellen)) nil!!)
    ))

(defun modify-patient-slot (pvar value) (*set pvar value))
```

The **in-error** function is in error because **foo-a!!** returns a pvar containing a copy of the data in **foo**'s slot, **a**. This pvar is allocated on the stack. The function **modify-foo-slot** then tries to ***set** this temporary pvar. Since the ***set** affects bits on the *Lisp stack, not bits in the **foo** pvar, the intended result is not obtained. Slot **a** of **foo** is never modified.

The **correct** function is correct because **alias!!** returns a pvar that points to the same bits in the Connection Machine as **foo**'s slot **a**. This aliased pvar can be meaningfully modified.

4.4 Miscellaneous Operations on Structure Pvars

The function **equalp!!** can be used on structure pvars to test for slot-by-slot equivalence. See section 7.1 for a complete definition of **equalp!!**.

structurep!! *any-pvar* [*Function*]

This function returns a boolean pvar with the value **t!!** if *any-pvar* is a structure pvar and **nil!!** if not.

!! *foo-object* [*Function*]

Pvars of some structure type, *foo-object-pvar*, may be constructed using the function **!!** on an object that is of Common Lisp structure type, *foo-object*. When this is done, every active processor receives an **equalp** copy of *foo-object*'s slot structure.

It is true that

```
(*all (equalp foo-object (pref (!! foo-object) 0)))
```

but it is never true that

```
(*all (eq foo-object (pref (!!  foo-object) 0)))
```

That is, replicating a front-end structure in all active processors using !! and then referencing the resultant structure pvar in any single processor, does *not* return the structure initially used **(eq)** but rather a copy **(equalp)** of that structure.

4.5 Scanning Structures

The only scan operation that can be used when scanning with a structure pvar is **copy!!**. (For a complete description of the function scan!!, see the **Lisp Reference Manual*, chapter 6.)

4.6 Detailed Documentation

The preceding descriptions of structure pvars is all the typical user of ***defstruct** will ever need to know. In what follows, further ***defstruct** options and slot options are explained. These options make it possible to change the names of the ***defstruct** accessor functions and the properties of the front-end **defstruct**. The options detailed here are equivalent to those provided in Common Lisp.

4.6.1 Options to *defstruct

:conc-name

The ***defstruct** option **:conc-name** takes one optional argument, *conc-name*, a symbol. The Connection Machine slot accessor function names are constructed by prefixing each slot name with the *conc-name* symbol and suffixing it with '!!'. The *conc-name* defaults to the name of the structure pvar suffixed with '-'. If **:conc-name** is specified as **nil**, then no *conc-name* is prepended to form the symbol name.

The **:conc-name** argument is also passed to the front-end **defstruct** as the option **:conc-name**.

Examples :

```
(*defstruct (foo) (slot 0.0 :type single-float))
```

defines the Connection Machine accessor **foo-slot!!**

```
(*defstruct (foo (:conc-name bar-))
  (slot 0.0 :type  single-float))
```

defines the Connection Machine accessor **bar-slot!!**

```
(*defstruct (foo (:conc-name nil))
  (slot 0.0 :type single-float))
```

defines the Connection Machine accessor **slot!!**

```
(*defstruct (foo (:conc-name))
  (slot 0.0 :type single-float))
```

defines the Connection Machine accessor **foo-slot!!**

:cm-constructor

If specified, **:cm-constructor** takes one argument, *cm-constructor*, a symbol. The *cm-constructor* symbol specifies the name of a function that creates Connection Machine parallel structures of type **(pvar** *struct-name***)**. If not specified, the Connection Machine constructor is formed by prepending 'make-' and appending '!!' to the structure pvar name.

Examples :

```
(*defstruct (foo)
   (slot 0.0 :type single-float))
```

defines the **make-foo!!** constructor

```
(*defstruct (foo (:cm-constructor make-boa!!))
   (slot 0.0 :type single-float))
```

defines the **make-boa!!** constructor

```
(*defstruct (foo (:cm-constructor))
   (slot 0.0 :type single-float))
```

defines the constructor **make-foo!!**

```
(*defstruct (foo (:cm-constructor nil))
   (slot 0.0 :type single-float))
```

does not define a constructor.

:constructor

If specified, **:constructor** takes one argument, *constructor*, a symbol. The *constructor* symbol specifies the name of a function that creates front-end instances of the structure *struct-name*. It is passed to the front-end **defstruct** as the **:constructor** option.

:copier

If specified, **:copier** is passed to the front-end **defstruct** as the **:copier** option. It takes one argument, a symbol.

:parallel-cm-predicate
:predicate

If specified, each of these takes one argument, a symbol.

The **:parallel-cm-predicate** option specifies the name of the predicate function defined by ***defstruct** for the structure pvar type. If no **:parallel-cm-predicate** option is specified, ***defstruct** defines a predicate with a default name, formed by appending "-p!!" to the name of the structure pvar type.

The **:predicate** option, if specified, is passed to **defstruct** as the Common Lisp **:predicate** option. It takes one argument, a symbol. If no **:predicate** option is specified, **defstruct** defines a predicate with the default name, formed by appending "-p" to the name of the front-end structure.

:include

If specified, the **:include** option takes one argument, a symbol. This symbol must be the name of a structure pvar type definition created by a previous invocation of ***defstruct**. The specified structure pvar type definition is included at the beginning of

the structure pvar type being defined by the current ***defstruct**. The resulting structure pvar type definition behaves as though all the slots of the included structure pvar type definition were specified textually before the slots of the new structure pvar. The accessors of the included structure pvar type correctly access the slots of structure pvars of the the new type. (See the more detailed description of this option is section 4.2, above.)

:print-function

This option is passed to **defstruct** as the **:print-function** option. The argument to **:print-function**, which must be a function name, is used to print the front-end structures.

:cm-uninitialized-p

This option is equivalent to providing the slot option **:cm-uninitialized-p** to every slot of the structure pvar type being defined.

4.6.2 *defstruct Slot Options

:type

This option is not optional. See the more detailed discussion in section 4.1.2, above for further information.

:cm-type

This option takes one argument, a valid *Lisp type specifier. It is an error if the *Lisp type specifier provided is not compatible with the **:type** option Lisp specifier. For instance, **(pvar boolean)** and **(unsigned-byte 16)** are not compatible, whereas **(pvar (unsigned-byte 16))** and **(member 0 1 2 3)** are.

:cm-initial-value

See the description of this option, in section 4.1.2, above.

:cm-uninitialized-p

See the description of this option, in section 4.1.2, above.

4.6.3 *defstruct Options Example

The code below defines two parallel structure types and illustrates the proper syntax to use for ***defstruct** options and slot options.

```
(*defstruct (foo )
   (a 3 :type (unsigned-byte 8))
   (b 0.0 :type single-float)
   )
```

This defines a structure pvar type **foo** with two slots, **a** and **b**, which may hold data of type **(unsigned-byte 8)** and **single-float**, respectively. Here is an example using the **foo** structure pvar type:

```
(*let ((my-foo (make-foo!! :a (!! 8) :b (!! 5.0))))
  (declare (type (pvar foo) my-foo))
  (frob my-foo))
```

A **foo** structure named **my-foo** is created and its slot values are initialized to values other than the the default initialization values.

The code below creates a pvar structure definition for **xyzzy**, illustrating the use of most of the ***defstruct** options and slot options.

```
(*defstruct (xyzzy
             (:conc-name plugh-)
             (:copier duplicate-xyzzy)
             (:cm-constructor create-xyzzy!!)
             (:constructor create-xyzzy)
             (:parallel-cm-predicate is-it-an-xyzzy?!!)
             (:predicate is-it-an-xyzzy?)
             (:print-function print-the-magic-word)
             (:include foo)
             )
  (c 3 :type (member 3 5) :cm-type (pvar (unsigned-byte 12))
       :cm-initial-value (random!! 10))
  (d nil :type boolean :cm-uninitialized-p t)
  )
```

Chapter 5

Virtual Processor Sets

The notion of virtual processors is unrelated to any construct or concept in Common Lisp. All operations and variables documented in this chapter are extensions to Common Lisp, designed to add power and flexibility to programs run on the Connection Machine system (CM).

The term *virtual processors* is used to indicate how many processors the CM logically operates, regardless of how many physical processors are contained in the machine. The number of virtual processors (VP's) in use at any given time is expressed in terms of dimensions, analogous to the dimensions of an array. The product of a series of virtual processor dimensions indicates how many virtual processors are operating when those dimensions are in effect. For instance, dimensions of (128 16 4) specify a machine configuration of 8192 VP's.

5.1 Virtual Processor Sets in Release 5.0

Version 5.0 provides a more efficient and flexible implementation of virtual processors than did previous releases. A new abstraction, termed *virtual processor sets*, allows the use of multiple sets virtual processor dimensions during a single session. At any given time after ***cold-boot** has been invoked, there is exactly one VP set active. The currently active VP set is known as the *current VP set*.

Prior to Version 5.0, it was possible to employ only one set of VP dimensions during any session. The dimensions argument to the ***cold-boot** function determined the number of virtual processors simulated by the CM until the next ***cold-boot**. This scheme proved inefficient when the processor requirements of program data varied. Users were forced to use the maximum number of virtual processors required by any data set a program used and to leave portions of the CM idle when smaller data sets were processed.

With Version 5.0, it is possible to specify a number of different virtual processor sets (VP sets), each defined by separate VP dimensions. Further, each VP set may have distinct pvars associated with it. Thus, data may be assigned to VP sets that are appropriately dimensioned. Previous to Version 5.0, only two-dimensional virtual processor grids were allowed. This restriction has been lifted: *n*-dimensional VP sets have been implemented.

Memory management of virtual processors has been improved with the introduction of virtual processor sets. Previously, the memory of each CM processor was divided into as many fixed-size segments as there were virtual processors. This scheme severely limited the amount of memory available, especially when large VP dimensions were used. The new scheme allows memory in both the *Lisp heap and the *Lisp stack to be allocated on an as-needed basis. (See the Paris documentation for a more detailed discussion of the changes to Connection Machine memory management introduced with Version 5.0.)

5.2 How Virtual Processor Sets Work

The default virtual processor set is defined at ***cold-boot** time. An argument to ***cold-boot**, called *initial-dimensions*, is a list of integers that define the number of virtual processors initially simulated by the CM.The product of the *initial-dimensions* integers is the number of virtual processors operated during the session by the Connection Machine system whenever no other virtual processor set is selected. For example,

```
(*cold-boot :initial-dimensions '(128 64 2))
```

configures the Connection Machine system to operate 16K virtual processors arranged in a cube.

In addition to the default VP set defined by ***cold-boot**, VP sets may be defined with the *Lisp **def-vp-set** and the **create-vp-set** operations. All that is fundamentally required to define a VP set is a choice of dimensions. This is illustrated by the following code.

```
(def-vp-set snap '(32 64))
(setq cake (create-vp-set '(128 256 1024))
```

Two VP sets are defined. The VP set **snap** has two dimensions; **cake** has three. **Snap** will persist after a ***cold-boot** while **cake** will disappear.

```
(*defvar snap-dragon (!! 120) "A flowering plant" snap)
(*defvar snapshot (!! 220) "A casual photograph" snap)
(*defvar cake-walk (!! 10) "An easy contest" cake)
```

Three pvars are created and associated with VP sets. Belonging to **snap** are the pvars **snap-dragon** and **snapshot**. Belonging to **cake** is the pvar **cake-walk**.

As the preceding example demonstrates, creating VP sets is often a simple process. Nonetheless, it can involve making subtle distinctions and using operations that have some complex features.

Virtual processor sets are created by being *defined* and *instantiated*. When a VP set is defined, *Lisp records the definition of that VP set. When pvars are associated with a VP set using ***defvar**, their definitions are stored with the VP set definition. When a VP set is instantiated, Connection Machine memory is allocated to store the pvars associated with that VP set. Definition and instantiation are not always concurrent. Prior to the first ***cold-boot**, it is legal to define a VP set but not to instantiate it.

Two *Lisp operations are provided specifically for creating VP sets: the **def-vp-set** macro and the **create-vp-set** function. The primary distinction between VP sets created by these operations is their longevity. VP sets created with **def-vp-set**, along with the pvars belonging to such VP sets, are generally reallocated every time ***cold-boot** is invoked. VP sets created with **create-vp-set**, along with the pvars belonging to such VP sets, are always destroyed every time ***cold-boot** is invoked.

Calls to **def-vp-set** may be made either before or after the first ***cold-boot** time. A call to **def-vp-set** defines a VP set, either completely or partially. If a call to **def-vp-set** is made after loading *Lisp and before calling ***cold-boot**, and if the definition is complete, then the defined VP set is allocated at the next ***cold-boot**. If a call to **def-vp-set** is made after calling ***cold-boot** and if that call completely specifies a VP set, then that VP set is defined and allocated all at once: when the **def-vp-set** form is evaluated.

If a VP set is not completely specified by **def-vp-set**, then the function **allocate-processors-for-vp-set** must be called sometime after calling ***cold-boot**. In this case, **allocate-processors-for-vp-set** completes the VP set definition and instantiates the VP set.

Calls to **create-vp-set** are always made after ***cold-boot**. VP sets created with this form are immediately defined and allocated.

While a VP set may have any number of pvars associated with it, a pvar may only belong to a single VP set. A pvar may be associated with a particular VP set by one of three methods. First, invocations of the **def-vp-set** form can include pvar definitions. Second, the function ***defvar** takes an optional *vp-set* argument. Third, from within a

with-vp-set** form, the functions **allocate!!**, ***let**, and ***let may be used to allocate pvars belonging to the current VP set.

A VP set is said to have a *geometry*. Generally, a list of dimensions sufficiently specifies the geometry of a VP set. However, the function **create-geometry** is provided to allow more complex control over the mapping of virtual processors onto physical processors. Such control is can significantly speed interprocessor communications.

There is always a current VP set. A *Lisp program may use multiple VP sets on the CM by switching between VP sets. Two *Lisp constructs change the current VP set: **set-vp-set** and ***with-vp-set**.

Without changing the current VP set, processors in the current VP set may send or fetch data from processors in another VP set. In general, however, most *Lisp operations require their pvar arguments to belong to the current VP set.

All the VP set functionality described above is detailed in this chapter, with the exception of communication between VP sets. Inter-VP-set communication is discussed in this manual supplement in chapter 6, entitled "*N*-Dimensional Interprocessor Communication in *Lisp."

5.3 Global Variables Related to VP Sets

default-vp-set [*Variable*]

This defines the current VP set when no other VP set is current. This variable is bound at ***cold-boot** time and always has as many virtual processors as were specified by the **:initial-dimensions** argument to the function ***cold-boot**.

For example, given a 16K CM, if ***cold-boot** is called with **:initial-dimensions** (256 256), then the dimensions of ***default-vp-set*** would be (256 256) and each physical processor would simulate four virtual processors; there would be 64K virtual processors.

If no initial dimensions are specified the first time ***cold-boot** is called, ***default-vp-set*** defaults to a two-dimensional VP set with virtual processors equal to physical processors. Thus, on a 16K CM, calling ***cold-boot** with no **:initial-dimensions** argument binds ***default-vp-set*** to a VP set with dimensions (128 128) and 16K virtual processors. If **:initial-dimensions** is not specified for subsequent calls to ***cold-boot**, the previous value of **:initial-dimensions** is used.

minimum-size-for-vp-set [*Variable*]

This defines the smallest number of virtual processors with which a VP set may be dimensioned. The product of the dimensions for any VP set must be greater than or equal to the value of this variable. Currently, this variable is equal to the physical size of the machine.

Before the first ***cold-boot**, it is an error to access this parameter.

current-vp-set [*Variable*]

This defines *Lisp's currently active VP set and defaults to ***default-vp-set***.

current-cm-configuration [*Variable*]

This variable is bound to a list of integers that define the dimensions of the current VP set.

Before the first ***cold-boot** it is an error to access this parameter.

number-of-dimensions [*Variable*]

This defines the number of dimensions in the current VP set. It is the rank of the current machine configuration. Before the first ***cold-boot** it is an error to access this parameter.

number-of-processors-limit [*Variable*]

This defines the number of virtual processors in the current VP set. Before the first ***cold-boot** it is an error to access this parameter.

current-send-address-length [*Variable*]

This defines the number of bits needed to hold the send address of a single processor within the current VP set. This parameter has the value **(integer-length (1- *number-of-processors-limit*))**.

Before the first ***cold-boot**, it is an error to access this parameter.

current-grid-address-lengths [*Variable*]

This variable is bound to a list of integers such that the *j*th element of the list defines the number of bits necessary to hold a grid address component for the *j*th dimension of the current VP set.

Before the first ***cold-boot**, it is an error to access this parameter.

The address length of a single grid dimension may be obtained by calling the function **dimension-address-length.**

dimension-address-length *dimension* [*Function*]

This function returns the number of bits necessary to represent a grid address coordinate for the specified dimension. This is simply the *n*th element of the list ***current-grid-address-lengths***.

The argument *dimension* must be between 0 and one less than the rank of the current VP set.

5.4 Operations to Create, Destroy, and Reinitialize Virtual Processor Sets

The *Lisp operations that create VP sets are: ***cold-boot, def-vp-set, create-vp-set, allocate-processors-for-vp-set,** and **let-vp-set.**

The *Lisp operations that destroy VP sets are **deallocate-vp-set** and ***cold-boot.**

The *Lisp operations that reinitialize VP sets are ***warm-boot** and ***cold-boot.**

This section contains definitions for each of these operations.

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

This operation should be called after loading in the *Lisp software and before attempting to execute *Lisp code. It resets the internal state of the *Lisp system and of the Connection Machine hardware.

The **:safety** keyword argument specifies a value for the *Lisp variable *interpreter-safety*. For a detailed discussion of interpreter safety see section 7.10.

The keyword arguments **:initial-dimensions** and :initial-geometry-definition specify the geometry of the initial VP set bound to ***default-vp-set***. One or the other but not both of these keyword arguments may be provided.

If supplied, *initial-dimensions* must be a list of integers, each of which is a power of 2. These are the dimensions of ***default-vp-set***. The product of the dimensions may be equal to the physical machine size. If not, the product of the dimensions must be a power of two multiple of the physical machine size. If unsupplied, *initial-dimensions* defaults to a list of two integers whose product is the physical machine size.

If supplied, *initial-geometry-definition* must be a geometry object. See the definition of **create-geometry**, below, for more details.

A successful ***cold-boot** returns two values: the number of physical processors and ***current-cm-configuration***.

A ***cold-boot** invocation performs the following operations:

- evaluates forms defined on the ***before-*cold-boot-initializations*** list
- destroys any pvars produced by **allocate!!** in a previous program execution
- destroys any VP sets produced by **create-vp-set** in a previous program execution
- attempts to attach the Connection Machine hardware if it not already attached and, if successful, calls the Paris function **cm:cold-boot**
- sets the value of the variable ***interpreter-safety***
- instantiates the VP set bound to ***default-vp-set***

  If *initial-dimensions* or *initial-geometry-definition* are provided, this information is the geometry information used to define ***default-vp-set***.

  If a previous ***cold-boot** has been done, and if no geometry information is provided, the previous geometry information from the previous ***cold-boot** is used.

  If a no ***cold-boot** has been done since the last **cm:attach**, and if no geometry information is provided, a suitable two-dimensional grid, based on the size of the physical machine, is chosen as the geometry for ***default-vp-set***.

- instantiates, using **def-vp-set** and in some arbitrary order, all defined VP sets that have geometry information
- allocates and initializes, using ***defvar** and in some arbitrary order, all pvars defined to belong to VP sets with geometries
- selects the VP set ***default-vp-set***, making it the ***current-vp-set***
- evaluates the forms defined on the ***after-*cold-boot-initializations*** list

def-vp-set *vp-set-name vp-set-dimensions* **&key** [*Macro*]
:geometry-definition-form
:*defvars

This defines a VP set named *vp-set-name* and returns the symbol, **set-name**. The **def-vp-set** macro should only be used at top-level.

The argument *vp-set-name* must be a symbol; it is bound globally to the VP set defined.

The *vp-set-dimensions* argument must be a quoted list of positive integers, a form that evaluates to a list of positive integers, or **nil**. If an argument is supplied to the keyword **:geometry-definition-form**, the *vp-set-dimensions* argument must be **nil**. If not **nil**, *vp-set-dimensions* specifies an *n*-dimensional array of virtual processors.

Each dimension must be a power of two. The product of all dimensions must be a power-of-two multiple of the physical machine size or equal to the physical machine size. The total size specified by *vp-set-dimensions* must be at least as large as ***minimum-size-for-vp-set***.

The argument to **:geometry-definition-form** must be a form which, when evaluated, produces a geometry object. Eval is applied to this form. If *geometry-definition-form* is provided, it incorporates information about the dimensions of the VP set being defined. Examples of appropriate forms are: a call to **create-geometry**, a symbol bound to the result of a call to **create-geometry**, and a user-defined form that evaluates to a geometry object. (See section 5.5, below, for a discussion of geometry objects.)

If either *vp-set-dimensions* or *geometry-definition-form* is supplied, the VP set *vp-set-name* is initialized and allocated at ***cold-boot** time. If either *vp-set-dimensions* or *geometry-definition-form* is supplied and a ***cold-boot** has already been executed, the VP set *vp-set-name* is initialized and allocated immediately.

If *vp-set-dimensions* is **nil** and the *geometry-definition-form* parameter is unsupplied or **nil**, then the VP set is defined but it is not allocated. Instead, the VP set may be

allocated on demand by calling the function **allocate-processors-for-vp-set** any time after calling ***cold-boot**. (See the definition of **allocate-processors-for-vp-set**, below.)

The keyword **:*defvars** takes a list of lists, each of which specifies a pvar belonging to the VP set *set-name*. Each sublist must have the format:

(*symbol* **&optional** *initial-value-form documentation pvar-type*)

Here, *symbol* is bound to a pvar with initial value *initial-value-form*, documentation *documentation*, and type *pvar-type*.

For each such sublist, if *pvar-type* is not **nil**, a form with the following construction is evaluated.

'(***proclaim** '(*type ,pvar-type ,symbol*))

Whether or not *pvar-type* is **nil**, the following form is evaluated.

'(***defvar** *,symbol ,initial-value-form ,documentation vp-set*)

where *vp-set* is the symbol *vp-set-name* given as the first argument to **def-vp-set**.

The **:*defvars** keyword provides the ability to textually associate pvars with their VP sets. Note that pvars thus specified are allocated and initialized when the VP set *set-name* is allocated. Such pvars are allocated and reinitialized upon ***cold-boot** invocation.

Examples:

```
(def-vp-set fred '(1024 32 128))

(def-vp-set george (list *number-of-processors-limit* 32))

(def-vp-set anne '(512 512) :*defvars  ((x (!! 1) nil
                                           (field-pvar 2))
                                       (y (self-address!!))))
```

create-vp-set *dimensions* **&key :geometry** [*Function*]

This function is used to define a VP set during program execution. It is an error to invoke **create-vp-set** prior to the first ***cold-boot**. Any VP set allocated using **create-vp-set** will be destroyed with the next ***cold-boot**.

The return value of **create-vp-set** is a front-end VP set structure.

The *dimensions* argument must be a list of positive integers or **nil**. If a list is supplied, each integer in the list must be an integer power of two and the product of all the integers in the list must be at least as large as ***minimum-size-for-vp-set***. If larger than the physical machine size, the product of all dimensions must be a power-of-two multiple of the physical machine size. The dimensions argument must be **nil** if an argument is supplied to the keyword **:geometry**. If not **nil**, dimensions logically specifies an *n*-dimensional array of virtual processors.

The argument to **:geometry** must be a geometry object obtained by calling **create-geometry**. If *geometry* is provided, it incorporates information about the dimensions of the VP set being defined. (See the definition of **create-geometry**, below, for more details.)

Examples:

```
(setq y (create-vp-set '(512 8 32))
(setq x (create-vp-set (append (vp-set-dimensions y) '(2 2))))
```

Two VP sets are created, a 3-dimensional configuration and a 5-dimensional configuration.

allocate-processors-for-vp-set *vp-set dimensions* **&key :geometry** [*Function*]

This function is used during program execution to instantiate a VP set which has been previously partially defined by **def-vp-set** without supplying a geometry. By omitting the geometry from a **def-vp-set** call and later calling **allocate-processors-for-vp-set**, it is possible to create VP sets with dimensions and geometries determined at run time. For instance, VP set geometries might depend on characteristics of data that are read off disk during program execution.

It is an error to invoke **allocate-processors-for-vp-set** before ***cold-boot** has been invoked.

The argument *vp-set* must be a VP set defined by a call to the **def-vp-set** macro in which the *set-dimensions* argument was **nil** and the **:geometry-definition-form** keyword argument was either **nil** or unsupplied.

The *dimensions* argument must be a list of integers or **nil**. If a list of integers is supplied, each integer must be a power of 2. The product of the dimensions must be at least as large as ***minimum-size-for-vp-set*** and, if larger than the physical machine size, a power-of-two multiple of the physical machine size. Such a list specifies the dimensions of a virtual array of processors named *vp-set*. The *dimensions* argument must be **nil** if an argument is supplied to the keyword **:geometry**.

If the keyword **:geometry** is supplied an argument, it must be a geometry object. If *geometry* is provided, it incorporates information about the dimensions of the VP set being defined. (A geometry object may be obtained by calling the function **create-geometry**. See the definition of **create-geometry**, below, for more details.)

Example:

```
(def-vp-set disk-data
  nil
  :*defvars
  ((disk-data-pvar nil nil '(pvar single-float)))
  )


(defun top-level-function (diskfile)
  (*cold-boot)
  (let ((number-of-elements
         (read-number-of-elements-from-disk diskfile)))
    (allocate-processors-for-vp-set disk-data
       (list (next-power-of-two->= number-of-elements)))
    (let ((array-of-data (read-data-from-disk diskfile)))
      (array-to-pvar array-of-data disk-data-pvar
           :send-address-end number-of-elements)
      (process-disk-data-in-cm disk-data)
     )))
```

let-vp-set (*vp-set-name vp-set-creation-form*) **&body** *body* [*Function*]

This macro creates a temporary VP set which may be used only within the body of the form. The VP set *vp-set-name* is bound to the value of *vp-set-creation-form*. The *body* forms are then executed. Finally—within an unwind-protect—**deallocate-vp-set** is called on the value of *vp-set-name* and the form is exited.

The argument *vp-set-name* must be a symbol. The argument *vp-set-creation-form* must be an executable form that includes a call to **create-vp-set**.

The forms of *body* may be any legal, executable forms.

The return value of **let-vp-set** is the value of the last form in the body.

Example:

```
(progn
 (let-vp-set (temp-cube (create-vp-set '(16 32 1024)))
   (*with-vp-set temp-cube
     (*let ((thoughts (!! 5))
            (random (random!! (!! 10)))
            )
       (declare (type (pvar integer) thoughts random))
       (do-something-with-temp-cube-vp-set random thoughts)
       )))
 (format t "Now the temp-cube vp-set no longer exists")
 )
```

Notice that the temporary VP set created by a **let-vp-set** form must be explicitly selected with a ***with-vp-set** form before it is used. Notice also that the **temp-cube** VP set is deallocated upon exit of the let-vp-set.

deallocate-vp-set *vp-set* [*Function*]

This function destroys the VP set *vp-set* regardless of whether it was created by a call to **def-vp-set** or to **create-vp-set**. All pvars belonging to *vp-set* are deallocated and destroyed. If *vp-set* was created with **def-vp-set**, then the symbol that names the VP set is made unbound.

***warm-boot** [*Macro*]

This macro resets some but not all internal *Lisp and Connection Machine states.

A call to ***warm-boot** must be made whenever a *Lisp program terminates abnormally. It is wise to call ***warm-boot** at the beginning of major entry points of *Lisp software applications.

No parameters are given to ***warm-boot**. The return value of a successful ***warm-boot** is **nil**.

Executing ***warm-boot** has the following effects:

- All allocated VP sets are restored to a state such that all their processors are active
- The ***current-vp-set*** is made to be the ***default-vp-set***
- All pvars allocated on the stack (i.e., any not created by **allocate!!** or ***defvar**) are removed from the stack and the stack pointer is reset.

5.5 The Geometry of Virtual Processor Sets

Every virtual processor set must have a geometry. A VP set has geometry in the sense that every VP set is defined by a front-end structure that includes a geometry object. A geometry object is itself a front-end structure. Often a geometry object consists of little more than a set of dimensions. A geometry object is automatically created by *Lisp when a call to ***cold-boot**, **def-vp-set**, **create-vp-set**, or **allocate-processors-for-vp-set** supplies an optional dimensions argument. Alternatively, a geometry object may be explicitly created by calling the function **create-geometry**. Such a geometry object may optionally be supplied to one of the operations that create VP sets.

create-geometry &key :dimensions :weights :orderings [*Function*]
:on-chip-bits :on-chip-pos
:off-chip-bits :off-chip-pos

This function returns a structure known as a geometry object. (See the definitions of ***cold-boot**, **def-vp-set**, **create-vp-set**, and **allocate-processors-for-vp-set** for discussions on how to use geometry objects.)

Specifying a **:dimensions** keyword argument is mandatory. The value of the **:dimensions** keyword must be a list of integers, each of which must be a power of 2. These dimensions define an *n*-torus that describes the shape of a virtual processor set. The product of the dimensions must be a power of two multiple of the physical machine size.

The remaining keyword arguments default to reasonable values if not specified. These arguments impact only the run-time performance of near neighbor communication and of certain non-local communication patterns. They do not affect functionality in any way.

If supplied, the value of **:weights** must be a list of integers, one for each dimension. This argument specifies the relative frequency of use for near neighbor communication in each dimension with respect to the other dimensions. Given the specified weighting, the Connection Machine system distributes data for optimal performance.

For example, specify *weights* as '(1 2 1) if, within a three-dimensional VP set, near neighbor communication is estimated to be twice as frequent in dimension 1 as in either dimension 0 or 2.

If a **:weights** value is supplied, *none* of the "chip-bits" values should be supplied. If a **:weights** value is not supplied, *all* of the "chip-bits" values should be supplied.

If supplied, the value of **:orderings** must be a list of symbols, one for each dimension. Only the symbols **:news-order** and **:send-order** may appear in the *orderings* list. If not supplied, *orderings* defaults to a list in which each member is the symbol **:news-order**. For those dimensions corresponding to **:news-order** ordering values, send addresses are gray-coded and mapped into NEWS addresses. For those dimensions corresponding to **:send-order** orderings values, no special address translation is done. Thus, it is possible to optimize a VP set geometry for near neighbor communication along certain dimensions and for general router communication along other dimensions. For more information on the effects of each type of ordering, see the *Paris Reference Manual*, Version 5.0.

The majority of *Lisp users will never need to use the "chip-bits" arguments; the *weights* argument is usually sufficient.

The **:on-chip-bits**, **:on-chip-pos**, **:off-chip-bits**, and **:off-chip-pos** arguments together specify a series of bitmasks that map send addresses into NEWS addresses. This can be useful if maximum control over interprocessor communication patterns at the hardware level is desired. These arguments are provided in *Lisp as a direct hook into Paris. For more information on how to design bitmasks for these arguments, see the *Paris Reference Manual*, Version 5.0.

The **create-geometry** function is designed to provide *Lisp users with a great deal of control over internal interprocessor communication speed within the context of a particular VP set. This can be useful, for instance, when it is critical to optimize the performance of scan operations.

5.6 Selecting a VP Set

When ***cold-boot** is called, ***default-vp-set*** becomes the currently selected VP set. Thereafter, provided that further VP sets are defined and allocated, it is possible to select alternate VP sets. The currently selected VP set is dynamically scoped. Two *Lisp forms are provided to change the currently selected VP set during a session: **set-vp-set**, and ***with-vp-set**.

set-vp-set *vp-set* [*Function*]

This function changes the currently selected VP set to *vp-set*.

The argument *vp-set* must be a VP set that is both defined and allocated.

The return value of a call to **set-vp-set** is *vp-set*.

***with-vp-set** *vp-set* **&body** *body* [*Macro*]

This macro encloses code which is executed in the context of the VP set *vp-set*.

The argument *vp-set* must be a VP set that is both defined and allocated.

The forms of *body* may be any legal, executable forms.

The currently selected VP set is dynamically scoped. The ***with-vp-set** form changes the currently selected VP set to *vp-set*. Thus, while a ***with-vp-set** form is executing, the global variables related to VP sets are dynamically bound to reflect the VP set context. The following global variables are affected when the current VP set is changed:

current-cm-configuration
current-vp-set
number-of-processors-limit
current-send-address-length
ppp-default-end
t!!
number-of-dimensions
log-number-of-processors-limit
current-grid-address-lengths
ppp-default-start
nil!!

The currently selected set (CSS) of processors active for a VP set is dynamically scoped. If execution passes from within a ***with-vp-set** form that reduces a VP set's CSS into code that selects another VP set, the CSS of the originally selected VP set is not automatically restored to include all processors. Thus, as execution moves between VP sets, each set maintains the state established for it immediately prior to selection of another VP set. This is illustrated by the example shown below.

Example:

```
(def-vp-set fred '(1024 32) :*defvars ((p nil nil (field-pvar 32))))
(def-vp-set anne '(512 512) :*defvars ((x (!! 1) nil(field-pvar 16))
                                       (y (self-address!!))))


(*with-vp-set fred                                    ;32,768 VP's
  (*when (evenp!! self-address!!))                    ;16,384 VP's
    (*with-vp-set anne                                ;262,144 VP's
      (*set x (-!! y x))
      (*with-vp-set fred                              ;16,384 VP's
        (*when (not!! (zerop!! (self-address!!)))
          (*set p (!! 1))                             ;16,383 VP's
          )
          (*set p (!! 1))                             ;16,384 VP's
          )))
  (*sum (!! 1)))                                      ;=> 32,768
```

When a VP set is created, it is defined to have all processors selected. The initial invocation to ***with-vp-set fred** activates all virtual processors defined by **fred**. Next, a ***when** statement reduces the number of **fred**'s active VP's by half. Notice that, even after a visit with VP set **anne**, the second invocation of ***with-vp-set fred** enters the **fred** VP set context as it was last encountered: with only processors of even-numbered addresses active. The ***when** form that follows further reduces **fred**'s context by deactivating processor 0. Inside this ***when** statement, `(*sum (!! 1))` returns 16,383 — the number of active VP's now in **fred**'s css. Finally, the nested ***with-vp-set fred** form is closed and execution passes back into the ***with-vp-set** form that originally selected **fred**. Now all of **fred**'s processors are again active and `(*sum (!! 1))` returns their count, 32,768.

5.7 Pvars Associated with VP Sets

A VP set may have several pvars associated with it. However, a pvar may be associated with only one VP set. The relationship between a VP set and its associated pvars is discussed in terms of ownership. Thus, a pvar is said to "belong to a VP set" and that VP set is said to "own the pvar."

Pvars may be associated with VP sets in several ways. The easiest method is to define pvars using **def-vp-set**, as described above. A pvar may also be defined and associated with a VP set by using the macro ***defvar** and providing the optional VP set pa-

rameter. Finally, from within a ***with-vp-set** form, pvars created by either **allocate!!**, ***let**, or ***let*** belong to the current VP set. The function **pvar-vp-set** is useful for verifying to which VP set a pvar belongs.

***defvar** *symbol* **&optional** *initial-value documentation vp-set* [*Macro*]

This macro defines a pvar named *symbol* and *symbol* is proclaimed special. The return value of a successful call to ***defvar** is *symbol*. The macro ***defvar** may be invoked either before or after ***cold-boot**.

Pvars created with ***defvar** are reallocated and reinitialized with every subsequent ***cold-boot**.

If a ***cold-boot** has been done, *symbol* is bound to Connection Machine memory allocated for the pvar. The pvar so allocated belongs to the VP set *vp-set* and has initial value *initial-value*.

If a ***cold-boot** has not been done, the definition of *symbol* as a pvar is stored on the front end and, at ***cold-boot** time, the pvar is allocated and initialized on the Connection Machine system.

If the VP set *vp-set* is explicitly deallocated by a call to **deallocate-vp-set**, the pvar to which *symbol* is bound is deallocated and symbol becomes unbound.

The optional argument *initial-value* must be a form that evaluates to a pvar. The *initial-value* form is evaluated in the context of *vp-set*. If *initial-value* is **nil** or is unsupplied, the initial value of *symbol* is indeterminate.

The optional *documentation* argument defaults to **nil**. If supplied, *documentation* must be a string enclosed in a pair of double quotes. Documentation of type variable is thus defined for *symbol*.

The optional argument *vp-set* defaults to ***default-vp-set***. If supplied, this must be the name of an existing vp-set.

Note that a pvar associated with a VP set is allocated when that VP set is allocated.

Example:

```
(def-vp-set census '(128 128))
(*defvar new-england-census nil nil census)
(*defvar mid-west-census nil nil census)
(*defvar southern-census nil nil census)
(*defvar west-coast-census nil nil census)

(*cold-boot)
```

Memory is allocated for the four regional census pvars.

```
(*with-vp-set census
    (setq cambridge-census (allocate!!))
    (setq berkeley-census (allocate!!)))
```

Memory is allocated for the two college town census pvars.

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

The regional census pvars exist but the college town pvars no longer exist (i.e., there is no longer any college town data in the CM). Referring to the value of either **cambridge-census** or **berkeley-census** is now an error.

Six pvars are associated with the VP set **census**. The four pvars named after regions of the United States are reallocated and reinitialized after every invocation of ***cold-boot**. The two pvars named after college towns disappear when ***cold-boot** is subsequently called. Note that it is an error to call **allocate!!** prior to the initial ***cold-boot**.

Note: To get rid of pvars created with ***defvar**, use the function ***deallocate-*defvars**.

allocate!! **&optional** *initial-value name type-declaration* [*Function*]
let**(*{*symbol* **&optional** *pvar*})** **&body** *body* [*Macro*]
let(*{*symbol* **&optional** *pvar*}***)** **&body** *body* [*Macro*]

These forms may be used from within a ***with-vp-set** form to allocate pvars belonging to the current VP set. For detailed descriptions of these operations, see the **Lisp Reference Manual*, Version 4.0, chapter 3.

Example:

```
(def-vp-set seafaring '(256 256))
(*with-vp-set seafaring
   (allocate!! (!!0) tug (pvar unsigned-byte 16))
   (*let ((schooner (!! 10)) (brigit (!! 5)))
      (*if (>!! (self-address!!) schooner)
        (*set tug schooner)
        (*set tug brigit)
   )
   tug))
```

pvar-vp-set *pvar* [*Function*]

This function returns the VP set to which *pvar* belongs.

The argument *pvar* may be any pvar.

5.8 Getting Information About a VP Set

vp-set-dimensions *vp-set* [*Function*]

This function returns a list of integers specifying the dimensions of VP set *vp-set*.

The *vp-set* argument must be a defined, allocated VP set.

describe-vp-set *vp-set* **:*defvars :verbose** [*Function*]

This function prints information about *vp-set*. The information displayed by **describe-vp-set** is derived from the front-end VP set structure created when *vp-set* was defined.

Executed for side effect, **describe-vp-set** returns **nil**.

The argument *vp-set* must be a VP set that has been defined. If *vp-set* has not been allocated, **describe-vp-set** will show most slot values as **nil**.

The keyword argument to :***defvars** must be a boolean. It defaults to **t**. If the default is used, pvars belonging to *vp-set* are described. Otherwise, pvars belonging to *vp-set* are not described.

The keyword argument to :**verbose** must be a boolean. It defaults to **nil**. If the default is used, only the most generally useful information is printed when **describe-vp-set** is invoked. If :**verbose** is **t**, additional information, such as the length of the grid address for each dimension, is printed.

A sample call to **describe-vp-set** are shown below.

```
(describe-vp-set *current-vp-set*)

  vp set name: *default-vp-set*
  geometry allocation form: nil
  dimensions: (32 32)
  geometry-id: 1
  nesting-level: 1
  *defvars belonging to *default-vp-set*
    name: a-foo, initial-value-form: (*lisp-i:make-foo!!),
    type: (pvar (structure foo))
    name: cube-temp, initial-value-form: (!! 0),
    type: (pvar (unsigned-byte *current-send-address-length*))
nil
```

In this example, ***current-vp-set*** is examined and discovered to be ***default-vp-set***, a two-dimensional VP set with two associated pvars, **a-foo** and **cube-temp**. The **geometry-id** is a unique number identifying the geometry of this VP set. The nesting-level is the number of nested ***with-vp-set** forms currently in effect for this VP set.

```
(describe-vp-set *default-vp-set* :verbose t)

  vp set name: *default-vp-set*
  geometry allocation form: nil
  dimensions: (32 32)
  geometry-id: 1
  nesting-level: 1
  paris vp id: 1
  geometry rank: 2
  grid-address-lengths: (5 5)
  *defvars belonging to *default-vp-set*
    name: foo, initial-value-form: (!! 2),
    type: nil
    name: cube-temp, initial-value-form: (!! 0),
    type: (pvar (unsigned-byte *current-send-address-length*))
nil
```

Here, ***default-vp-set*** is described in more depth by supplying a **:verbose** value of **t**. The grid-address-lengths list is the value to which ***current-grid-address-lengths*** is bound when this VP set is the currently selected VP set.

Chapter 6

N-Dimensional Interprocessor Communication

This chapter describes new *Lisp functionality for interprocessor communication. In previous releases, *Lisp only worked with two–dimensional grids of virtual processors. Version 5.0 introduces *Lisp support for *n*-dimensional grids of virtual processors, where *n* is any positive integer less than or equal to 31.

The introduction of *n*-dimensional processor configurations affects the functionality of interproccessor communication operations and is concomitant with the newly introduced virtual processor set (VP set) abstraction. Most *Lisp communication operations have been enhanced to use *n* dimensions. Also, many new operations have been added to support communication between VP sets in *n* dimensions.

The *Lisp operations supporting interprocessor communication allow processor addressing between virtual processors and across virtual processor sets. It is not necessary to deliberately use VP sets in order to use *n*-dimensional interprocessor communication. This is true because the *n*-dimensional grid specified at ***cold-boot** time may be sufficient throughout the session.

Operation definitions and descriptions in this chapter use the terms processor and virtual processor interchangeably. The term machine configuration is used to mean the geometry of the current virtual processor set. (For a discussion of virtual processors and virtual processor sets in *Lisp, see chapter 5 of this supplement.)

6.1 Global Variables Related to *N*-Dimensional Communication

The following global variables are often used with *n*-dimensional communication operations in *Lisp:

current-cm-configuration
number-of-dimensions
number-of-processors-limit
current-send-address-length
current-grid-address-lengths

For definitions of each of these, see section 5.3, "Global Variables Related to VP Sets," in the chapter 5 of this supplement.

6.2 Enhanced *Lisp Communication Operations

The following functions have been enhanced to work with *n* dimensions:

***cold-boot**	**scan!!**
dimension-size	**self-address-grid!!**
array-to-pvar-grid	**pvar-to-array-grid**
cube-from-grid-address	**cube-from-grid-address!!**
grid-from-cube-address	**grid-from-cube-address!!**
off-grid-border-p!!	**off-grid-border-relative-p!!**

Unless otherwise noted, this section includes a full definition for each of these operations. These definitions supersede the definitions in previous manuals. (See section 6.4 ,entitled "Communication Across Virtual Processor Sets," for information about new communication operations.)

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

This operation must be called after loading in *Lisp and before attempting to execute *Lisp code. It resets the internal state of the *Lisp system and of the Connection Machine hardware.

The **:safety** keyword argument specifies a value for the *Lisp variable *interpreter-safety*. For a detailed discussion of interpreter safety see section 7.10.

Either the **:initial-dimensions** or the **:initial-geometry-definition** keyword argument—but not both—may be used to specify the geometry of the initial Connection Machine configuration in *n* dimensions. Whichever of these keyword argument is supplied to ***cold-boot**, defines the ***default-vp-set*** and its geometry. For example:

```
(*cold-boot :initial-dimensions '(64 64))
(*cold-boot :initial-dimensions '(64 64 32))
(*cold-boot :initial-dimensions '(2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2))
(*cold-boot :initial-geometry-definition
  '(create-geometry :dimensions (5 36 2) :weights (2 1 3)))
```

For a complete definition of ***cold-boot**, see chapter 5 of this supplement, section 5.4.

scan!! *pvar function* **&key :direction :segment-pvar :include-self :dimension** [*Function*]

A detailed definition of the **scan!!** operation appears in the **Lisp Reference Manual*, revised for Version 5.0, chapter 6. The **:dimension** keyword argument is new with *Lisp Version 5.0, as is the ability to scan pvars defined in *n* dimensions.

The **:dimension** keyword value defaults to **nil**, indicating that the scan is performed in send address order. Alternatively, *dimension* may be given as an integer between 0 and one less than the rank of the current VP set. If *dimension* is an integer value, the scan operation is performed along that dimension. If desired, *dimension* may be specified as **:x**, **:y**, or **:z**; these are equivalent to dimensions 0, 1, and 2.

For example,

```
(scan!! pvar 'copy!! :dimension :z)
```

copies the value of each point in the x,y plane at $z=0$ into the corresponding point in the x, y plane at $z=1$, and thence to x, y at $z=2$, and so on to $z=n$, where n is the extent of z.

Note: The new **:dimension** keyword argument allows **scan!!** to replace **scan-grid!!**. Therefore, **scan-grid!!** is now obsolete.

dimension-size *dimension* [*Function*]

This function returns the size of the specified dimension in the context of the current machine configuration. The return value is the maximum allowable grid address for the dimension, plus one.

The *dimension* argument can be any non-negative integer less than the rank of the current machine configuration.

self-address-grid!! *dimension-pvar* [*Function*]

For each processor in the currently selected set, this function calculates the grid address of that processor along the specified dimension.

The *dimension-pvar* argument must be a pvar containing a non-negative integer in each processor. Each of these integers must be less than the rank of the current machine configuration.

The return value of **self-address-grid!!** is a pvar containing a non-negative integer in each processor. In each processor, the return value is the component of that processor's grid address that corresponds to the local value of *dimension-pvar*.

cube-from-grid-address *integer* **&rest** *integers* [*Function*]

This function translates a series of integers specifying the grid address of a single processor into a single integer specifying the send address of that processor.

Each argument specifies a coordinate point along one axis in an *n*-dimensional grid. At least one argument is required and the number of integer values supplied must equal the rank of the current machine configuration.

The argument *integer* must be a non-negative integer. The *integers* arguments are a series of non-negative integers.

The return value of **cube-from-grid-address** is an integer. This is a front-end scalar value.

For example, assuming a three-dimensional configuration is in effect:

```
(cube-from-grid-address 10 20 30) => 1036
```

Here, the processor located at coordinates (10, 20, 30) is discovered to have a send address of 1036. Note that this value is not predictable from the information given in

the example; in addition to the grid address values, it depends on the geometry of the current VP set, on the number of physical processors, and on the system software version in use.

cube-from-grid-address!! *integer-pvar* [*Function*]
&rest *integer-pvars*

This function translates a series of pvars, specifying the grid addresses of some number of processors, into a single pvar that specifies the send addresses of those processors. This is the parallel equivalent of **cube-from-grid-address**.

The argument *integer-pvar* must be a pvar containing a non-negative integer in each processor. The *integer-pvars* arguments are a series of pvars, each containing non-negative integers.

Each integer pvar argument contains, in each processor, a coordinate point along one axis in the current *n*-dimensional geometry. At least one argument pvar is required and the number of integer pvars supplied must equal the rank of the current machine configuration.

The return value of **cube-from-grid-address** is an integer pvar.

For example, assuming a cube configuration is in effect:

```
(cube-from-grid-address!! (!! 10) (!! 20) (!! 30))
=> (!! 1036)
```

Here, the send address of the processor located at coordinates (10, 20, 30), 1036, is stored in all active processors. Notice that this is a somewhat simplified example in that each argument specifies an identical value in each processor. Also be aware that **(!! 1036)** is not a predictable value, given the information presented in the example; in addition to the grid address values, it depends on the geometry of the current VP set, on the physical size of the machine, and on the version of system software in use.

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

This function translates a single integer representing the send address of a single processor into a single integer representing the grid address of that processor along the specified dimension.

The *cube-address* argument must be a non-negative integer within the current machine configuration's range of send addresses. This range extends from zero through **(1- *number-of-processors-limit*)**, inclusive.

The *dimension* argument must be a non-negative integer between zero and one less than the rank of the current machine configuration.

The return value of **grid-from-cube-address** is a non-negative integer: one of the integers in the series of integers that constitute the complete grid address of the specified processor. This is a front-end scalar value.

For example, assume a four-dimensional machine configuration and assume that the processor referenced by send address 6534 has a grid address of (6 52 75 259).

```
(grid-from-cube-address 6534 2) => 75
```

Here, the grid address component corresponding to dimension 2 is returned. To obtain all the grid address components, call **grid-from-cube-address** repeatedly, specifying a different dimension each time.

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

In each processor, this function translates the cube address specified that processor's value of *cube-address-pvar* into a corresponding grid address along the dimension specified by the local value of *dimension-pvar*. This is the parallel equivalent of **grid-from-cube-address**.

The *cube-address-pvar* argument must be pvar containing a non-negative integer in each processor. Each of these integers must be within the range zero through **(1- *number-of-processors-limit*)**, inclusive.

The *dimension-pvar* argument must be a pvar containing, in each processor, a non-negative integer between zero and the rank of the current machine configuration minus one.

The return value of **grid-from-cube-address!!** is an integer pvar containing non-negative integers. In each processor the integer returned is the *dimension-pvar* grid address component of the processor referenced by *cube-address-pvar*.

Notice that each argument pvar may contain different values in each processor, as may the pvar returned by the operation.

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

This function tests grid addresses for validity. In each processor, the grid address tested is the integer series constituted by that processor's values of the argument pvars. This function determines whether or not these grid addresses point within the bounds defined by the current VP set.

All arguments must be integer pvars. The number of arguments must be equal to the rank of the current machine configuration.

The return value of **off-grid-border-p!!** is a boolean pvar. It contains **t** in each processor in which *integer-pvar(s)* specify an invalid grid address. In all other processors, **nil** is returned.

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

This function tests relative grid addresses for validity. In each processor, a relative address is formed by the integer series defined by that processor's values of the pvar arguments. Within this series, the *j*th integer specifies the distance, along the *j*th dimension, between the current processor and the processor referenced. This function determines whether or not the relative grid address in each processor, when added to the self addresses, points within the bounds of the current machine configuration.

All arguments must be integer pvars. The number of arguments must be equal to the rank of the current machine configuration.

The return value of **off-grid-border-p!!** is a boolean pvar. It contains **t** in each processor in which the *integer-pvar(s)* specify an invalid relative grid address. In all other processors, **nil** is returned.

6.3 New *Lisp Communication Operations

Three new operations, **news!!**, **news-border!!**, and ***news** have been added to *Lisp to support general *n*-dimensional NEWS communication. Also, the new communication functions **spread!!**, and **reduce-and-spread!!** are provided to extend the **scan!!** class of functions.

news!! *pvar* **&rest** *integers* [*Function*]

This function does near neighbor fetch communication. Each processor in the currently selected set reads the value of *pvar* from the processor that is *integers* processors away in the *n*-dimensional grid of the current VP set.

The return value of a call to **news!!** is a pvar of the same type as *pvar* containing, in each active processor, the value fetched by that processor.

The argument *pvar* may be any pvar or expression that evaluates to a pvar. The *pvar* argument is evaluated in the context of the processors specified by *integers*. This is especially important if *pvar* is an expression. A *pvar* expression is evaluated in the context of the processors that send data back to the processors in the currently selected set.

Each **&rest** argument must be a non-negative integer and the number of *integers* supplied must equal the rank of the current VP set. If the value of the *j*th **&rest** argument in active processor A is *n*, then *n* specifies the grid address of processor B relative to A. B is *n* processors away from that A along the *j*th dimension.

There is no upper bound on the integers arguments. If the relative addressing specified by *integers* results in addresses that would otherwise read off the edge of the current machine configuration, those addresses wrap around to the other side.

The function **news!!** is in some respects a replacement for the obsolete function **pref-grid-relative!!**. Notice, however, that **news!!** wraps addresses and that it only allows relative grid addressing in which a consistent distance is spanned between all active processors and the processors from which they read.

For example, given a two-dimensional grid configuration,

```
(*set dest-pvar (news!! source-pvar 1 0))
```

causes each selected processor to read the value of **source-pvar** from the processor 1 step away in grid address space. Notice that *source-pvar* need not be in the CSS. Similarly, given a three-dimensional configuration,

```
(*set dest-pvar (news!! source-pvar 1 8 2))
```

causes each selected processor to read the value of **source-pvar** from the processor that is **(1, 8, 2)** steps away in grid address space.

news-border!! *pvar border-pvar* **&rest** *integers* [*Function*]

This function replaces the obsolete function **pref-grid-relative!!** in cases where **pref-grid-relative!!** was given a **:border-pvar** argument.

```
(news-border!! pvar border-pvar 1 1)
<=>
(pref-grid-relative!! pvar (!! 1) (!! 1) :border-pvar border-pvar)
```

***news** *source-pvar dest-pvar* **&rest** *relative-coordinate-integers* [*Macro*]

This function does near neighbor store communication. Each processor in the CSS of the current VP set takes the value of *source-pvar* and stores it in *dest-pvar*, in the processor that is *relative-coordinate-integers* away along each dimension.

The return value of a ***news** form is **nil**; ***news** is executed for side effect.

The argument *source-pvar* may be any pvar or expression that evaluates to a pvar. The *source-pvar* argument is evaluated in the context of the CSS.

The argument *dest-pvar* may be any pvar or expression that evaluates to a pvar. The *dest-pvar* argument is evaluated in the context of the processors specified by *relative-coordinate-integers*. This is especially important if *dest-pvar* is an expression. A *dest-pvar* expression is evaluated in the context of the processors that receive data from processors in the currently selected set.

Each **&rest** argument must be a non-negative integer. The number of *relative-coordinate-integer* arguments must equal the rank of the current VP set. If the value of the *j*th **&rest** argument in active processor A is *n*, then *n* specifies the grid address of processor B relative to A. B is *n* processors away from that A along the *j*th dimension.

There is no upper bound on the values of the *relative-coordinate-integer* arguments. If the relative addressing specified results in addresses that would otherwise read off the edge of the current machine configuration, those addresses wrap around to the other side.

Example:

```
(*news (!! 3) foo 1 1)
```

This puts the value 3 into every processor which is 1 processor east and 1 processor south of each active processor. The value is stored into the pvar foo.

Notice that ***news** is to **news!!** as ***pset** is to **pref!!**. Thus, while **news!!** retrieves information from nearby processors, ***news** sends information to nearby processors. Like **news!!**, ***news** wraps around the grid. As with ***pset**, the processors receiving data during a ***news** operation need not be active.

spread!! *any-pvar dimension coordinate* [*Function*]

This function spreads data across the Connection Machine processors along dimension *dimension*. The data to be spread is taken from the *coordinate* processor along the dimension *dimension*. The data is spread to all the active processors. Thus, operating on a 2-dimensional pvar, for instance, it is possible to spread the data in any given column or row across the rows or columns.

The first argument, *any-pvar*, may be any pvar. The argument *dimension* must be a non-negative integer scalar within the range of ***number-of-dimensions***, or **nil**. If *dimension* is **nil**, the spread is done using send addressing and *coordinate* specifies a send address.

The argument *coordinate* must be a non-negative integer scalar. It is an error if *coordinate* specifies any processors outside the currently selected set.

The return value of a call to **spread!!** is a pvar of the same type as *any-pvar*.

reduce-and-spread!! *any-pvar scan-operator* **&key :dimension** [*Function*]

Conceptually, this function first performs a

```
(scan!! any-pvar scan-operator :dimension dimension)
```

It then takes the **scan!!** result from the last active processor along the scanning dimension and performs a backwards **copy!!** scan. A pvar containing the result of this copy scan is returned. Thus, the **scan!!** results are spread to all the processors which participated in the **reduce-and-spread!!**.

The first argument, *any-pvar*, may be any pvar. The second argument, *scan-operator*, may be any binary operator.

The **:dimension** keyword argument, *dimension-scalar*, must be a non-negative integer scalar within the range of *any-pvar*'s dimensions, or **nil**. If *dimension* is **nil**, cube scanning is done.

This function is provided because it may be significantly faster to use it than to do a **scan!!** followed by a reverse copy scan.

6.4 Communication Across Virtual Processor Sets

From within the context of one VP set, it is possible to write data to and read data from pvars belonging to another VP set. This section details the *Lisp operations used to effect such inter-vp-set communications. (For a detailed discussion of VP sets and the operations used to create them, see chapter 5 of this manual supplement.)

6.4.1 Addresses Translation Across VP Sets

When operating in a single VP set, it is often necessary to convert grid addresses to send addresses. The following functions do such translations:

cube-from-grid-address	**cube-from-grid-address!!**
grid-from-cube-address	**grid-from-cube-address!!**

These functions are documented in section 6.2, above. They compute addresses correctly only for the current VP set.

To compute an address in another VP-set the following functions are used:

cube-from-vp-grid-address	**cube-from-vp-grid-address!!**
grid-from-vp-cube-address	**grid-from-vp-cube-address!!**

These functions are different from other address translation functions in one significant regard: they take a VP set argument specifying the VP set of the source address, in which the translation is to be done.

It is important to remember that only these '-vp-' functions can be used across VP sets. The address translation function that do not include '-vp-' in their names may not be used across VP sets. The reason for this is that different VP sets may have different geometries, even if they have the same dimensions.

cube-from-vp-grid-address *vp-set integer* **&rest** *integers* [*Function*]

This function translates a series of integers specifying the grid address of a single processor in *vp-set* into a single integer specifying the send address of that processor in the context of the VP set *vp-set*.

The argument *vp-set* must previously have been both defined and instantiated.

The *integer* argument must be a non-negative integer. It specifies a coordinate point along the 0th dimension in *vp-set*'s geometry.

The **&rest** integers arguments must be non-negative integers. Each **&rest** argument specifies a coordinate point along one axis in the *n*-dimensional grid of *vp-set*'s geometry. The number of integer arguments supplied, including the second required parameter and the **&rest** arguments, must equal the number of dimensions in *vp-set*.

The return value of **cube-from-vp-grid-address** is an integer: the send address corresponding to the specified *vp-set* grid address.

Consider the following example, assuming **my-vp** has a three-dimensional geometry.

```
*number-of-dimensions* => 2
*current-vp-set* => your-vp
(cube-from-vp-grid-address my-vp 10 20 30) => 1036
```

Although the current VP set is **your-vp**, the processor located at coordinates (10, 20, 30) in **my-vp** may be referenced, without changing the VP set context, by the send address 1036, using, for example, the function **pref**. (Note that the value 1036 is used here solely for the purpose of illustration; actual send addresses vary depending on the current VP set geometry, on the number of physical processors, and on the system software version in use.)

cube-from-vp-grid-address!! *vp-set integer-pvar* **&rest** *integer-pvars* [*Function*]

This function translates a series of pvars, specifying the grid addresses of some number of processors in *vp-set*, into a single pvar that specifies the send addresses of those processors. This is the parallel equivalent of **cube-from-vp-grid-address**.

The *vp-set* argument must be a VP set that has previously been both defined and instantiated.

The *integer-pvar* argument must be a non-negative integer pvar. In each processor, it specifies a coordinate point along the 0th dimension in *vp-set*'s geometry.

The **&rest** integers arguments must be non-negative integers. The **&rest** arguments specify coordinate points along consecutive axes in the *n*-dimensional grid of *vp-set*'s geometry. The number of integer pvar arguments supplied, including the second re-

quired parameter and the **&rest** arguments, must equal the number of dimensions in *vp-set*.

The return value of **cube-from-vp-grid-address** is an integer pvar.

For example, assuming **my-vp** has a three-dimensional geometry, the following call might be made.

```
(cube-from-vp-grid-address!! my-vp (!! 10) (!! 20) (!! 30))
=> (!! 1036)
```

Here, the send address equivalent of the **my-vp** set cube address (10, 20, 30) is discovered to be 1036; a copy of this value is stored in all active processors. (Note that the value 1036 is used here solely for the purpose of illustration; actual send addresses vary depending on the geometry of the current VP set, on the number of physical processors, and on the system software version in use.)

grid-from-vp-cube-address *vp-set cube-address dimension* [*Function*]

This function translates a single integer, representing the send address of a single processor in *vp-set*, into a single integer representing the grid address of that processor along the specified dimension in VP set *vp-set*.

The *cube-address* argument must be a non-negative integer within *vp-set*'s range of send addresses.

The *dimension* argument must be a non-negative integer between zero and one less than the rank of *vp-set*'s dimensions.

The return value of **grid-from-vp-cube-address** is a non-negative integer: one of the integers in the series of integers that constitute the complete grid address of the specified processor, in the context of a non-current VP set. This is a front-end scalar value.

For example, assume **my-vp** has a four-dimensional geometry and assume that the processor referenced by send address 6534 is, in the geometry of **my-vp**, a grid address of (6 52 75 259).

```
(grid-from-vp-cube-address my-vp 6534 2) => 75
```

Here, the grid address component corresponding to dimension 2 is returned. To obtain all the grid address components, call **grid-from-vp-cube-address** repeatedly, specifying a different dimension each time.

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

In each processor, this function considers the cube address specified by that processor's value of *cube-address-pvar* to be in the context of *vp-set*. It translates this value into a grid address and returns the coordinate value of the dimension specified by *dimension-pvar*. This is the parallel equivalent of **grid-from-vp-cube-address**.

The *cube-address-pvar* argument must be a pvar containing a non-negative integer in each processor. Each of these integers must be within the range of valid send addresses for *vp-set*.

The *dimension-pvar* argument must be a pvar containing, in each processor, a non-negative integer between zero and the rank of *vp-set*'s dimensions minus one.

The return value of **grid-from-vp-cube-address!!** is an integer pvar containing non-negative integers. In each processor the integer returned is the *dimension-pvar* grid address component, in the geometry of *vp-set*, of the processor referenced by *cube-address-pvar* in the context of *vp-set*.

off-vp-grid-border-p!! *vp-set integer-pvar* **&rest** *integer-pvars* [*Function*]

This function is similar to **off-grid-border-p!!**. It tests grid addresses for validity relative to a specified VP set.

The number of *integer-pvar* arguments must be equal to the number of dimensions in the *vp-set* argument.

The return value of **off-vp-grid-border-p!!** is a boolean pvar. It contains **t** in each processor for which the local values of the *integer-pvar(s)* specify an invalid grid address. In all other processors, **nil** is returned.

6.4.2 Address Translation Examples

Below are some examples of address translation across VP sets.

```
(def-vp-set 1d-vp-set '(8192))
(def-vp-set 2d-vp-set '(128 128)
            :*defvars '((self-x (self-address-grid!! (!! 0)))
                        (self-y (self-address-grid!! (!! 1)))))
```

```
(*with-vp-set 1d-vp-set
  (pref self-x (cube-from-vp-grid-address 2d-vp-set 3 4)))
=> 3


(*with-vp-set 1d-vp-set
  (pref self-y (cube-from-vp-grid-address 2d-vp-set 3 4)))
=> 4


(*with-vp-set 1d-vp-set
  (grid-from-vp-cube-address!!
    2d-vp-set (cube-from-vp-grid-address!! 2d-vp-set (!! 3) (!! 4))
      (!! 0)))
=> (!! 3)


(*with-vp-set 1d-vp-set
   (grid-from-vp-cube-address!!
    2d-vp-set (cube-from-vp-grid-address!! 2d-vp-set (!! 3) (!! 4))
      (!! 1)))
=> (!! 4)
```

Two VP sets are defined, each with a different geometry. **1d-vp-set** is one-dimensional; **2d-vp-set** is two-dimensional. **2d-vp-set** has two pvars, **self-x** and **self-y**, and they are initialized with the integers 0 and 1 in each processor, respectively.

In the context of **1d-vp-set**, the send address of the processor at **2d-vp-set** coordinates (3, 4) is calculated and and its value of **self-x** fetched. Next the **self-y** value in the same processor is fetched. Next, the inverse relationship between grid and send addresses is demonstrated by taking partial grid addresses of send addresses of grid addresses. The originally-supplied coordinates are returned.

Note: In these examples the use of ***with-vp-set** is unnecessary. It is used solely to illustrate how a VP set address translation function can be called from within the context of a VP set other than the one passed to the function.

6.4.3 Inter-VP Set Communication Operations

To transmit data between processors referenced by pvars belonging to different VP sets, four operations may be used: ***pset**, **pref!!**, **pref** and **setf** of **pref**. Some of these operations have changed with Version 5.0. The ***pset** operation now operates across VP sets without requiring an extra VP set argument. The **pref!!** and **pref** operations

now take optional VP set keyword arguments. The **pref** operation now evaluates its arguments the same way **pref!!** evaluates its arguments.

NOTE

The arguments to ***pset** have changed. The optional *notify* and *collision-mode* arguments have become keyword arguments.

***pset** *combiner value-pvar dest-pvar cube-address-pvar* [*Macro*]
&key :notify :collision-mode :vp-set

This operation copies the value of *value-pvar* from each processor in the currently selected set and writes it as the value of *dest-pvar* in each processor referenced by *cube-address-pvar*.

The arguments *value-pvar* and *cube-address-pvar* are evaluated in the context of the CSS of the current VP set. These arguments must be pvars belonging to the current VP set.

The *value-pvar* may be any pvar containing elements that can legally be copied into *dest-pvar*.

The *dest-pvar* argument may be any pvar in any VP set; it does not need to belong to the current VP set.

The *cube-address-pvar* may contain integer values that constitute valid cube addresses for the VP set to which *dest-pvar* belongs. Alternatively, an address object pvar may be used as the value of *cube-address-pvar*. (See section 6.5 for a discussion of address objects.)

If supplied, the **:vp-set** keyword argument must be the name of the VP set to which *dest-pvar* belongs. This argument is available solely for optimization. If a *vp-set* argument is not supplied, *Lisp determines the VP set to which *dest-pvar* belongs.

The return value of ***pset** is **nil**; ***pset** is executed for side effect.

The value of the *combiner* argument determines how multiple source values are combined if directed to a single destination processor. (For a further description of ***pset**

and the possible values that may be specified for the *combiner* argument as well as for the notify and collision modes, see the **Lisp Reference Manual*, revised for Version 5.0, chapter 6.)

pref!! *pvar-expression cube-address-pvar* [*Macro*]
&key :collision-mode :vp-set

The **pref!!** operation returns a pvar containing the value of *pvar-expression* obtained from the processors addressed by *cube-address-pvar*.

The sequence and context in which **pref!!** evaluates its arguments is somewhat unusual. First, **pref!!** evaluates *cube-address-pvar*. Then, in the VP set to which *pvar-expression* belongs, exactly those processors referenced by *cube-address-pvar* are selected. In those processors, *pvar-expression* is evaluated.

The argument *pvar-expression* may textually be a symbol that evaluates to a pvar or an expression that evaluates to a pvar. The *pvar-expression* pvar may belong to any VP set; it need not belong to the current VP set. However, if it is an expression rather than a symbol, and if it evaluates to a pvar in a VP set other than the current VP set, then a **:vp-set** argument is required.

The *cube-address-pvar* may contain integer values that constitute valid cube addresses for the VP set to which *pvar-expression* belongs. Alternatively, an address object pvar may be used as the value of *cube-address-pvar*. (See section 6.5 for a discussion of address objects.) The processors pointed to by *cube-address-pvar* do not need to be part of the currently selected set for the source VP set.

If supplied, the *vp-set* argument must evaluate to the VP set to which the pvar *pvar-expression* belongs. If an expression is specified textually as *pvar-expression*, and if the value of *pvar-expression* belongs to a different VP set, then a **:vp-set** value must be specified.

The value of the **:collision-mode** keyword argument determines what happens if more than one processor attempts to read from a single processor during a **pref!!** operation. The possible values for this argument are: **nil**, **:no-collisions**, **:collisions-allowed**, and **:many-collisions**. The default *collision-mode* value is **nil**. The default, **:collision-mode nil**, invokes a Paris instruction (**CM:GET-IL**), which uses the CM-2 backward routing hardware. As the number of collisions increases, this tends to be faster than **:collisions-allowed** and **:many-collisions**, but it uses much more temporary memory.

If a **pref!!** form causes *no* collisions, specify **:collision-mode** as **:no-collisions**. If there are *few* collisions, specify **:collisions-allowed** or use the default, **nil**. If there are *many* collisions, specify **:many-collisions** or use the default, **nil**. These last choices must be

made heuristically. While the *collision-mode* default, **nil**, is faster than **:collisions-allowed** or **:many-collisions** when there are many collisions, it uses more memory. Try using the default. If the program runs out of memory, change the *collision-mode* for the offending **pref!!** form(s) to **:many-collisions** or **:collisions-allowed**.

NOTE ABOUT PORTING FROM 4.3 TO 5.0

Existing code that specified no *collision-mode* argument to **pref!!** and thereby relied on the old *collision-mode* default, **:collisions-allowed**, will now get the new default, **nil**. Due to this change in the semantics of **pref!!**, *collision-mode* specifications in existing 4.3 *Lisp code should be reconsidered.

pref *pvar-expression cube-address* **&key :vp-set** [*Macro*]

The **pref** operation returns a Lisp value obtained by evaluating *pvar-expression* in the processor addressed by *cube-address*.

The sequence and context in which **pref** evaluates its arguments is somewhat unusual. First, *cube-address* is evaluated. Second, in the VP set to which *pvar-expression* belongs, the lone processor referenced by *cube-address* is selected. Finally, in that processor, *pvar-expression* is evaluated. The result is returned to the front end.

The argument *pvar-expression* may textually be a symbol that evaluates to a pvar or an expression that evaluates to a pvar. The *pvar-expression* pvar may belong to any VP set; it need not belong to the current VP set. If it is a symbol, no **:vp-set** keyword argument is required. Conversely, if *pvar-expression* is an expression and if it evaluates to a pvar in a VP set other than the current VP set, a value must be specified for **:vp-set**.

If supplied, the *vp-set* argument must evaluate to the VP set to which *pvar-expression* belongs. If a *vp-set* argument is not specified, *pvar-expression* is assumed to belong to the current VP set.

If *pvar-expression* is an expression, then, for the duration of the evaluation of *pvar-expression* only, the CSS will be set to the one processor from which **pref** is reading. This includes switching VP sets, which may be necessary if the **:vp-set** argument is specified.

6.4.4 Inter-VP Set Communication Examples

To communicate across VP sets, simply specify a pvar in another VP set and make sure the specified send address is valid for that VP set. The data will be transmitted to the specified pvar and processors. Examples using ***pset** and **pref!!** are given below. For convenience, the same VP sets are used in each example.

```
(def-vp-set 1d-vp-set '(8192))
(def-vp-set 3d-vp-set '(128 128 4)
            :*defvars '((3d-self    (self-address!!))
                        (3d-self-x (self-address-grid!! (!! 0)))
                        (3d-self-y (self-address-grid!! (!! 1)))
                        (3d-self-z (self-address-grid!! (!! 2)))))

(*with-vp-set 1d-vp-set
  (*let ((temp (pref!! 3d-self (random!! (!! 8192)))))
    (*and (and!!
            (=!! (grid-from-vp-cube-address!! 3d-vp-set temp (!! 0))
                 (pref!! 3d-self-x temp))
            (=!! (grid-from-vp-cube-address!! 3d-vp-set temp (!! 1))
                 (pref!! 3d-self-y temp))
            (=!! (grid-from-vp-cube-address!! 3d-vp-set temp (!! 2))
                 (pref!! 3d-self-z temp))))))
=>  t

(*with-vp-set 1d-vp-set
  (*let (temp)
    (*with-vp-set 3d-vp-set
      (*when (<!! (self-address!!) (!! 8192))
       (*pset :no-collisions (self-address-grid!! (!! 1))
              temp (self-address!!))))
    (*and
     (=!! (grid-from-vp-cube-address!! 3d-vp-set (self-address!!)
                                       (!! 1))
          temp))))
=>  t
```

Two VP sets are defined, one-dimensional **1d-vp-set** and three-dimensional **3d-vp-set**. **3d-vp-set** has four pvars, all each of which are initialized. Notice that **3d-self-x**, **-y**, and **-z** are each initialized to integer values in the grid address range of their VP set while **3d-self** holds, in each processor, the processor send address.

The two ***with-vp-set** forms illustrate how address translation, reading, and writing operations can be performed on one VP set from within the context of another.

The first ***with-vp-set** form demonstrates the correspondence between **grid-from-vp-cube-address!!** and **pref!!**. The grid address of each processor is compared with its contents.

The second ***with-vp-set** form illustrates in which VP set context ***pset** evaluates each of its arguments. Notice that **temp** is ***let** within **1d-vp-set**. It is then ***pset** to a grid address evaluated within the context of **3d-vp-set**. The call to **grid-from-vp-cube-address** emphasizes that **temp** was set to a **3d-vp-set** grid address.

The next example shows how **pref!!** evaluates its source pvar expression in the context of the processors from which it reads.

```
(*with-vp-set 1d-vp-set
  (*let ((temp (pref!! (self-address-grid!! (!! 0))
                       (self-address!!)
                       :vp-set 3d-vp-set)))
    (*and (=!! (grid-from-vp-cube-address!! 3d-vp-set
                                            (self-address!!)
                                            (!! 0))
               temp))))

=>  t


(*with-vp-set 1d-vp-set
  (*let ((temp (pref!! (self-address-grid!! (!! 0))
                       (self-address!!))))
                       ; missing :vp-set argument
    (*and (=!! (grid-from-vp-cube-address!! 3d-vp-set
                                            (self-address!!)
                                            (!! 0))
               temp))))

=>  nil
```

Notice that the the *vp-set* keyword argument to **pref!!** is not required. In both ***with-vp-set** forms, the *source-pvar* is an expression. If no **:vp-set** argument is given, the source pvar expression is evaluated in the CSS of the current VP set.

The macro **pref** is similar. Consider the following.

```
   (set-vp-set 1d-vp-set)
=> 1d-vp-set

   (pref 3d-self-y 26)
```

```
=> 1

   (pref (1+!! 3d-self-z) 563)
=> error
```

In the first **pref** form, the arguments are unambiguous. A call to **(pvar-vp-set 3d-self-y)** easily determines that **3d-self-y** belongs to **3d-vp-set**. On the other hand, the second **pref** form causes its first argument to be evaluated in the context of the current vp set, **1d-vp-set**. **(1+!! 3d-self-z)** is unknown in this context and an error results.

```
   (pref (1+!! 3d-self-z) 563 :vp-set 3d-vp-set)
=> 3
```

The above form specifies the VP set of the source expression and can therefore execute.

6.5 Address Objects—an Experimental Addressing Feature

A new approach to grid addressing is introduced with a feature known as *address objects*. Address objects simplify and generalize grid addressing across VP sets.

(pvar address-object) *[Pvar Type]*

Address objects are structures defined with ***defstruct**. They are defined to contain grid coordinates. The most salient feature of address objects is that the functions for creating them do not need to know which VP set is being referenced. *Lisp automatically computes this information when necessary.

*Lisp provides functions for creating and manipulating address objects. The address object creation functions **grid** and **grid!!** each take an arbitrary number of coordinate integers and return an address object containing those coordinates. Address object manipulation functions extract coordinates from address objects and increment specified dimensions of an address object.

As with other *Lisp addressing techniques, address objects provide two kinds of address values: front-end scalars and pvars. Accordingly, there are both scalar and parallel versions of the functions that operate on address objects. A scalar address object

is a single structure residing on the front end computer. A parallel address object is a parallel structure residing in a pvar on the Connection Machine system.

Address objects add little functionality to *Lisp. Most of the same results can be obtained with the operations **cube-from-vp-grid-address** and **cube-from-vp-grid-address!!**. For instance, address objects are intended to be passed as addresses to **pref!!**, **pref**, **setf** of **pref!!**, and **setf** of **pref**. This is also true of the results of **cube-from-vp-grid-address!!** and **cube-from-vp-grid-address**. Nonetheless, address objects can be more convenient and easier to use than these translation functions.

Address objects offer several other significant advantages in addition to convenience. An address object may be used to address any VP set within the address object's coordinate range. Also, address objects continue to point to the same processor in a VP set, even after that VP set has grown or shrunk. In contrast, **cube-from-vp-grid-address!!** would have to be reinvoked to retranslate grid addresses if the referenced VP set geometry changed. It is anticipated that VP sets with dynamic resizing will be implemented in the near future.

(See section 7.11 for a description of a debugging feature for use with address objects, **ppp-address-object**.)

grid &rest *integers* [*Function*]

This function creates and returns a front-end address object that contains the specified integers.

The *integers* argument must be a sequence of VP set dimension integers.

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

This function creates and returns a pvar of address objects containing the specified integer pvars.

grid-relative!! &rest *integer-pvars* [*Function*]

This function is equivalent to

```
(grid!! (+!! integer-pvar-0 (self-address-grid!! (!! 0)))
   (+!! integer-pvar-1 (self-address-grid!! (!! 1)))
   (+!! integer-pvar-2 (self-address-grid!! (!! 2)))
   ...)
```

address-nth *front-end-address-object dimension* [*Function*]

This function returns the specified grid coordinate of *front-end-address-object*. For example:

```
(address-nth (grid x y z) 0)  => x
(address-nth (grid x y z) 2)  => z
```

address-nth!! *address-object-pvar dimension-pvar* [*Function*]

This function returns the specified grid coordinate of a parallel address object *address-object-pvar*. For example:

```
(address-nth!! (grid!! x y z) (!! 1))  => y
(address-nth!! (grid!! x y z) (!! 2))  => z
```

address-rank *front-end-address-object* [*Function*]

This function returns the number of coordinates in *front-end-address-object*.

address-rank!! *address-object-pvar* [*Function*]

This function returns the number of coordinates in a parallel address object.

address-plus-nth *front-end-address-object integer dimension* [*Function*]

This function returns a front end address object that is a copy of *front-end-address-object* but with the specified dimension incremented by integer. For example:

```
(address-plus-nth (grid x y z) i 1)  => (grid x (+ i y) z)
(address-plus-nth (grid x y z) i 2)  => (grid x y (+ i z))
```

address-plus-nth!! *address-object-pvar integer-pvar dimension-pvar* [*Function*]

This function returns a pvar of address objects that is a copy of *address-object-pvar* but with the specified dimension incremented by integer-pvar. For example:

```
(address-plus-nth!! (grid!! x y z) i (!! 1))  => (grid!! x (+!! i
y) z)
(address-plus-nth!! (grid!! x y z) i (!! 2))  => (grid!! x y (+!!
i z))
```

self!! [*Function*]

This function returns an address object that contains the grid coordinates of each processor. It is equivalent to:

```
(grid!! (self-address-grid!! (!! 0))
   (self-address-grid!! (!! 1))
   ...
   (self-address-grid!! (!! n))
```

where *n* is **(1- *number-of-dimensions*)**.

Example using address objects:

```
(def-vp-set 2d-vp-set '(64 64)
  :*defvars ((a (!! 55))
             (2d-address-object (grid!! (!! 10) (!! 20)))))

(def-vp-set big-2d-vp-set '(256 256)
  :*defvars ((b (!! 66))))

(defun foo (pvar1 address)
  (setf (pref!! pvar1 address)
        (+!! (pref!! pvar1 address)
             (!! 5))))

(*with-vp-set 2d-vp-set
  (*when (=!! (self-address!!) (!! 0))  ; make sure only 1
                                        ; selected processor
    (foo a 2d-address-object)
    (foo b 2d-address-object))
```

```
(pref a (grid 0  0))   =>  55
(pref a (grid 10 20))  =>  60
(pref b (grid 0  0))   =>  66
(pref b (grid 10 20))  =>  71
```

6.5.1 What Address Objects Do

An address object is a defined with ***defstruct** and contains two slots, one for a send address and one for a geometry-id number. Given a geometry that contains a grid coordinate, any grid coordinate may be translated into a send address with that geometry. The send address may also be translated back to the original grid coordinate with that same geometry.

The functions **grid!!** and **grid** always translate the specified grid coordinate into a send address and a geometry-id that contains that grid coordinate.

When an address object is passed to **pref!!**, to ***pset**, or any of the communication functions, it is examined to make sure that the geometry-id is the same as that of the VP set with which the operation is communicating. If the geometry-id is not correct, the address object is translated back to its original grid coordinates and then retranslated to a send address for the correct geometry. This send address is then used by the communication function just as it would use any send address.

As a result of this automatic translation, an address object may be used to point to a grid address for any VP set within range.

When automatic translation occurs, the address object itself is modified. If the address objects is repeatedly used on the same VP set, translation overhead is incurred only once: the first time it is used. Subsequent uses involve no translation; the geometry-id of the address object matches the geometry-id of the VP set being used for communication.

6.6 Obsolete *Lisp Communication Functions

The following operations have become obsolete with the introduction of Version 5.0.

***pset-grid**	***pset-grid-relative**
pref-grid!!	**pref-grid-relative!!**
pref-grid	**scan-grid!!**

Alternatives to each obsolete operation are given below.

*pset-grid

Instead of ***pset-grid** use one of the following constructs.

```
(*setf (pref!! dest-pvar (grid!! x y)) value-pvar)

(*setf
 (pref!! dest-pvar (cube-from-grid-address!! x y))
 value-pvar)

(*pset :no-collisions value-pvar dest-pvar
  (grid!! x y))

(*pset :no-collisions value-pvar dest-pvar
  (cube-from-grid-address!! x y))
```

*pset-grid-relative

Instead of ***pset-grid-relative** use one of the following constructs.

```
(*setf (pref!! dest-pvar (grid-relative!! x y)) value-pvar)

(*setf (pref!! dest-pvar
         (cube-from-grid-address!!
           (+!! x (self-address-grid!! (!! 0)))
           (+!! y (self-address-grid!! (!! 1)))))
       value-pvar)
```

pref-grid!!

Instead of **pref-grid!!** use one of the following constructs.

Rather than: `(pref-grid!! source-pvar x y)`

do: `(pref!! source-pvar (grid!! x y))`

or: `(pref!! source-pvar (cube-from-grid-address!! x y))`

pref-grid-relative!!

Instead of **pref-grid-relative!!** use one of the following constructs.

Rather than:
```
(pref-grid-relative!! source-pvar x y)
```

do:
```
(pref!! source-pvar (grid-relative!! x y))
```

or:
```
(pref!! source-pvar (cube-from-grid-address!!
  (+!! x (self-address-grid!! (!! 0)))
  (+!! y (self-address-grid!! (!! 1)))))
```

Rather than:
```
(pref-grid-relative!! source-pvar (!! x) (!! y))
```

do:
```
(news!! source-pvar x y)
```

Read the definition of **news!!** in this chapter; these last two forms are not quite equivalent. In this case, **news!!** will not signal an error if a processor is reading off the edge of the VP set geometry configuration; **news!!** will wrap around to the other end of any dimension for which it is given an out-of-range index.

Rather than:
```
(pref-grid-relative!! source-pvar (!! x) (!! y)
                      :border-pvar foo)
```

do:
```
(news-border!! source-pvar foo x y)
```

See the definition of **news-border!!** in this chapter. The function **news-border!!** is similar to the obsolete **pref-grid-relative!!** when **pref-grid-relative!!** was given a **:border-pvar** argument.

pref-grid

Instead of **pref-grid** use one of the following constructs.

```
(pref source-pvar (grid x y))

(pref source-pvar (cube-from-grid-address x y))
```

scan-grid!!

Instead of **scan-grid!!** use **scan!!** with its new **:dimension** keyword argument.

Rather than: `(scan-grid!! 2D-pvar max!! :dimension :x)`

do: `(scan!! 2D-pvar max!! :dimension :x)`

Chapter 7

Assorted New *Lisp Features

7.1 Generally Useful Forms

help &optional *symbol* [*Function*]

When given no argument, **help** prints a message describing where to find information about *Lisp. When given a symbol defined by the *Lisp language, help prints information about the symbol, including whether it is a function, a macro, a *defun, or a variable, and whether the symbol is new as of Version 5.0.

describe-pvar *pvar* **&optional** *stream* [*Function*]

This function prints out information about a pvar in a neat format. The printed information includes length, type, VP set, and absolute memory address.

Examples:

```
(describe-pvar (!! 2))
=>
Pvar Name: nil
  Location: 4
  Field Id: 65536
  Length: 2
  Type: :field
  Vp Set Name: *default-vp-set*
  Vp Dimensions: (32 16)
  Constant value: 2

nil
```

allocated-pvar-p *pvar* [*Function*]

This function determines whether or not *pvar* has CM memory allocated for it. The return value of **allocated-pvar-p** is either **:stack**, **:heap** or **nil**. If its argument has been allocated on the *Lisp stack and has not been deallocated, **:stack** is returned. If its argument has been allocated on the *Lisp heap and has not been deallocated, **:heap** is returned. Otherwise. **nil** is returned.

Examples:

```
(allocated-pvar-p (!! 3)) => :stack
(allocated-pvar-p (allocate!! (!! 3))) => :heap

(setq x (!! 3)) => #<field-pvar 12-2>
(*warm-boot) => nil
(allocated-pvar-p x) => nil
(setq y (allocate!! (!! 2)))
=> #<field-pvar-* allocate!!-return 1336-2>
(*cold-boot) => 512
(32 16)
(allocated-pvar-p y) => nil
```

***setf** *place1 value1* **&optional** *place2 value2 ... place-n value-n* [*Macro*]

This operation takes one or more sets of place-value pairs and evaluates each argument. For each pair, it updates the pvar data found at the CM locations accessed by the value of *place-n* with the value of *value-n*. The return value of a ***setf** form is **nil**; ***setf** is executed for side effect.

This is the *Lisp equivalent of the Common Lisp **setf** macro. It should be used instead **setf** within a *Lisp expression. The use of **setf** within *Lisp expressions is obsolete with the release of Version 5.0.

The *place* arguments must be pvar accessor forms such as **aref!!**, **sideways-aref!!**, **pref**, **pref!!** and those constructed by ***defstruct**. Note that ***setf** will work on *place* arguments that return pvar accessors that are themselves pvars as well as on pvar accessors that are scalars.

The most common use of ***setf** is to change the value of pvar array elements and pvar structure slots. (See chapters 3 and 4 for descriptions of these pvar types.) For example,

```
(*setf (aref!! 3by6-array-pvar (!! 2) (!! 5)) (!! 28))
```

changes the value of element 2, 5 of **3by6-array-pvar** in each processor to 28.

```
(*setf (foo-struct-slot1!! foo-struct-pvar) (!! 84))
```

changes the value of **slot1** of the structure pvar **foo-struct-pvar** in each processor to 84.

***unless** *pvar* **&body** *body* [*Macro*]

This form subselects processors from the currently selected set (CSS). Within the body of the form, all active processors in which the value of *pvar* is **nil** are selected. Upon exit from a ***unless** form, the CSS is returned to the state that was in effect prior to the execution of the form. The ***unless** form is the same as the ***when** form except that the condition pvar *pvar* is negated. Thus:

```
(*unless unworthy-pvar body-forms)
<=>
(*when (not!! unworthy-pvar) body-forms)
```

***locally** *declaration-1 declaration-2 ... declaration-n* **&body** *body* [*Macro*]

This macro is used to provide declarations for the *Lisp compiler. The declarations *declaration-1* through *declaration-n* are used by the compiler for the body of the *body* form. A ***locally** declaration must be a **declare** form. Any valid compositions of **declare** may be used within a ***locally** form, including **optimize** and ***optimize** forms.

In previous releases, declarations would only be seen by the *Lisp compiler when used within a ***defun**, ***let**, or ***let*** form. With the use of ***locally**, the user is now able to give the *Lisp compiler type information and optimization directives anywhere in a program.

Examples:

```
(defun locally-test (j)
  (*compile ()
    (*locally
      (declare (type fixnum j))
      (*let (temp)
   (declare (type (unsigned-byte-pvar 32) temp))
   (*set temp (!! j))
   ))))
```

```
(defun *locally-example (result)
  (*locally
    (declare (type single-float-pvar result))
    (do-for-selected-processors (j)
      (*locally
        (declare (type fixnum j))
        (flet
          ((local-pvar-function (x)
             (*locally
               (declare (type single-float-pvar x result))
               (declare (*optimize (safety 0)))
               (*set result (+!! x (!! j)))
               )))
          (dotimes (i *current-cm-configuration*)
            (*locally
              (declare (type fixnum i))
              (*let ((temp (*!! (+!! (float!! (!! i)) (!! j))
                                (sin!! (!! j)))))
                (declare (type single-float-pvar temp))
                (local-pvar-function temp)
                ))))))))
```

Without ***locally**, the *Lisp compiler could handle the expressions in the above examples that include (!! j) only if each use of (!! j) were replace by (!! **(the fixnum j)**). In most cases, using ***locally** once within each enclosing form is easier.

Notice that ***locally** allows declaration of the arguments to local functions defined by **flet** and **labels**. Previously there was no way to do this.

power-of-two-p *positive-integer* [*Function*]

This function returns **t** if *positive-integer* is a power of two, otherwise it returns **nil**.

next-power-of-two->= *positive-integer* [*Function*]

This function returns an integer satisfying **power-of-two-p** and greater than or equal to *positive-integer.*

compare!! *numeric-pvar1 numeric-pvar2* [*Function*]

This function returns a pvar having values -1, 0, or 1, depending on whether its first argument is less than, equal to, or greater than, its second argument, respectively. The

arguments *numeric-pvar1* and *numeric-pvar2* must both be non-complex numeric pvars. A pvar of type **(pvar (signed-byte 2))** is returned.

return-pvar-p [*Declaration*]

The declaration forms

```
(declare (return-pvar-p t))
(declare (return-pvar-p nil))
```

are recognized by the *Lisp forms ***all**, ***when**, ***let**,***let*** and ***defun**.

A **return-pvar-p** declaration is a promise that the enclosing form either always or never returns a pvar. Using **return-pvar-p** declarations results in more efficient code, both interpreted and *compiled. Violating a **return-pvar-p** declaration is an error.

Examples:

```
(*when (<!! (self-address!!) (!! 10))
  ;; We promise that the *WHEN will not return a pvar
  (declare (return-pvar-p nil))
  ;; In fact it returns a number
  (*sum number-of-elements-pvar)
  )

(*let (x y z j)
  (declare (type single-float-pvar x y z))
  ;; We promise that the *LET will return a pvar
  (declare (return-pvar-p t))
  (declare (type (field-pvar 32) j))
  ;; In fact, +!! returns a pvar and thus the *LET returns a pvar
  (+!! (floor!! (min!! x y z)) j)
  )
```

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

This function returns an integer pvar of the same type and length as the argument. The result pvar contains a reverse copy of *integer-pvar*'s bits, so that the high-order bits become the low-order bits and vice versa.

The argument *integer-pvar* must be an integer pvar.

Note: This function relies on the internal representation of pvars in the Connection Machine system and therefore cannot work in the *Lisp simulator.

null!! *pvar* [*Function*]

This function is identical to **not!!**.

rem!! *numeric-pvar numeric-divisor-pvar* [*Function*]

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

7.2 Type Predication Functions

equalp!! *pvar1 pvar2* [*Function*]

This function is equivalent to **eql!!** if *pvar1* and *pvar2* are boolean pvars. It is equivalent to **char-equal!!** if they are character pvars. If *pvar1* and *pvar2* are numeric pvars, it is equivalent to **=!!**. If the parameters are structures or arrays, **equalp!!** returns the logical AND of calling itself on the slot pvars or element pvars, respectively, of the structures or arrays.

booleanp!! *pvar* [*Function*]

This predicate returns **t** in each processor in which *pvar* contains either **t** or **nil**, and returns **nil** in every other processor. When using general pvars, this can be useful to determine which processors contain boolean values.

Standard Common Lisp does not have a boolean type. *Lisp defines such a type as boolean < = > **(member t nil)**.

typep!! *pvar scalar-type* [*Function*]

This function is the parallel version of the Common Lisp function **typep**. It tests whether the value of *pvar* in each processor is of type *scalar-type*. The results of this predicate test are returned as a pvar containing **t** in each processor where *pvar* is of type *scalar-type* and containing **nil** elsewhere. For example:

```
(typep!! (!! t) 'boolean) => t!!
(typep!! (self-address!!) '(integer 0 10))
(typep!! (float!! (self-address!!)) '(float 0.0 10.0))
```

The last two invocations of **typep!!** above return **t** in processors 0 through 10 and **nil** elsewhere.

The argument *pvar* may be any pvar. The argument *scalar-type* must be one of the following type specifiers.

array	**bignum**	**bit**	**bit-vector**
boolean	**character**	**complex**	**complex**
double-float	**fixnum**	**float**	**front-end**
integer	**long-float**	**mod**	**nil**
null	**number**	**short-float**	**signed-byte**
single-float	**standard-char**	**string**	**string-char**
t	**unsigned-byte**	**vector**	

In addition, a user-defined structure type specifier may be used as the value of *scalar-type.*

Any of these valid type specifiers may be composed using **or**, **and**, **not**, and **member** in order to test *pvar* against more than one type.

Note: No *Lisp equivalent of the Common Lisp **satisfies** type constructor is provided.

7.3 Type Coercion and Conversion Functions

coerce!! *pvar pvar-type* [*Function*]

The **coerce!!** function is the parallel equivalent of the Common Lisp **coerce** function.This function attempts to convert *pvar* to type *pvar-type.* If this is possible, the result is returned as a new pvar allocated on the *Lisp stack. If *pvar* is already of type *pvar-type*, *pvar* is simply returned. If the specified conversion is not possible, an error is signaled.

The argument *pvar* may be any pvar. The argument *pvar-type* must be a valid *Lisp pvar type specifier.

It is not generally possible to convert any pvar to any type whatsoever; only certain conversions are permitted:

- An integer pvar (a signed-byte or unsigned-byte pvar) may be converted to an integer pvar type of a different size. For instance, a pvar of type **(pvar (unsigned-byte 8))** may be coerced to a pvar of type **(pvar (signed-byte 16))**.
- Integer pvars may be converted to floating-point pvar types. For example, a pvar of type **(unsigned-byte-pvar 16)** may be converted to a pvar of type **(pvar single-float)**.
- A floating-point pvar may be converted to a floating-point pvar of a different size. For instance, a pvar of type **(pvar single-float)** may be coerced to a pvar of type **(pvar double-float)**.
- An integer pvar or a float pvar may be converted to a complex pvar. For example, a single-float pvar can be converted to a complex pvar for which both exponent and significand are of type **(pvar double-float)**
- A complex pvar may be converted to a complex pvar of a different size. Thus, a pvar of type **single-complex-pvar** can be converted to a pvar of type **double-complex-pvar**.
- An integer pvar may be converted to a character pvar. This conversion is identical to that performed by the function **int-char!!**.
- A string-char array pvar of length 1 may be converted to a character pvar.
- Any pvar, except an array or a structure pvar. may be converted to a general pvar.
- An array pvar's element type may be converted in accordance with the permitted conversions mentioned above. For instance, an array pvar with elements of type single-float may be coerced to an array pvar with elements of type double-float.

Explicit single-argument type conversion functions may be used in place of **coerce!!**. Examples of *Lisp functions in this category are: **character!!**, **complex!!**, **float!!**, and **truncate!!**.

taken-as!! *pvar pvar-type* [*Function*]

This function is unlike any in Common Lisp. It is somewhat similar to the C language **cast** function in that it allows a pvar of one type to be used as though it were of another type. The function **taken-as!!** returns the original bits of *pvar* with type *pvar-type*. No coercion or change in representation occurs. For example,

```
(taken-as!! (!! 1.0) '(pvar (unsigned-byte 32))
=> (!! 1065353216)
```

The argument *pvar* may be any pvar, except a structure pvar. The argument *pvar-type* must be a valid *Lisp pvar type specifier with a length no larger that the length of *pvar*'s initial type.

Note: This function relies on the internal representation of pvars in the Connection Machine system and therefore cannot work in the *Lisp simulator.

Examples:

```
(taken-as!! (!! #C(1.0 1.0)) '(pvar (array single-float (2))))
```

This demonstrates that a complex pvar can be taken as a one-dimensional array pvar containing 2 single-float numbers in each processor.

```
(*proclaim '(type (pvar (unsigned-byte 8)) U8))
(*defvar U8)
(fun-that-requires-unsigned-byte-8 U8)
(fun-that-requires-bit-vector-8
  (taken-as!! U8 '(pvar (bit-vector 8))))
(fun-that-requires-unsigned-byte-8 U8)
```

Here, **U8** is an unsigned-byte pvar of length 8. The call to **taken-as!!** allows **U8** to be passed to a function that expects a bit-vector pvar of length 8.

ffloor!! *number-pvar* **&optional** *divisor-pvar* [*Function*]
fceiling!! *number-pvar* **&optional** *divisor-pvar* [*Function*]
ftruncate!! *number-pvar* **&optional** *divisor-pvar* [*Function*]
fround!! *number-pvar* **&optional** *divisor-pvar* [*Function*]

These functions are the parallel equivalents of the Common Lisp functions **ffloor**, **fceiling**, **ftruncate**, and **fround**. They behave like **floor!!**, **ceiling!!**, **truncate!!**, and **round!!**, except that the result in each processor is always a floating-point number rather than an integer. The argument pvars may contain either integers or floating-point numbers. The behavior is as if **ffloor!!**, for instance, gave its arguments to **floor!!** and then applied **float!!** to the result.

scale-float!! *float-pvar integer-pvar* [*Function*]

This function takes a floating-point pvar and an integer pvar; it returns, in each processor, that processor's *float-pvar* component multiplied by 2 to that processor's *integer-pvar* component power. For instance:

```
(scale-float!! (!! 3.5) (!! -1)) <=> (!! 1.75)
(scale-float!! (!! 1.0) (!! 2))  <=> (!! 4.0)
```

float-sign!! *float-pvar1* **&optional** *float-pvar2* [*Function*]

This function returns a floating-point pvar result with the same sign as *float-pvar1* and the same absolute value as *float-pvar2*. The argument *float-pvar2* defaults to a pvar of 1's; therefore, **(float-sign!! x)** always produces, in each processor, a 1.0 or a -1.0 with the same format as **x**.

7.4 Floating-Point Limits

Common Lisp provides a set of constants that define the largest and smallest floating-point representations provided by an implementation. (See *Common Lisp: The Language*, section 12.10) *Lisp makes this information available with a series of functions.

most-positive-float!! *floating-point-pvar* [*Function*]
least-positive-float!! *floating-point-pvar* [*Function*]
most-negative-float!! *floating-point-pvar* [*Function*]
least-negative-float!! *floating-point-pvar* [*Function*]

These functions each return a floating-point pvar with the same floating-point format as the argument *floating-point-pvar* but with a value of the named quantity. That is, the lengths of the return value's exponent and significand will be the same as those of *floating-point-pvar*. In each case, the argument *floating-point-pvar* may be any floating point pvar. For example,

```
(pref (most-positive-float!! (!! 0.0)) 0)
=> 3.4028235e38
```

The same result would be obtained with an argument of **(!! 5.8)** or with any single-precision floating-point pvar.

In each processor, the value returned by **most-positive-float!!** is the floating-point number closest to positive infinity that can be represented by the Connection Machine system (the CM) in the same floating-point format as *floating-point-pvar*.

In each processor, the value returned by **least-positive-float!!** is the positive floating point number closest to (but not equal to) zero that can be represented by the CM in the same floating-point format as *floating-point-pvar*.

In each processor, the value returned by **most-negative-float!!** is the floating point number closest to negative infinity that can be represented by the CM in the same floating-point format as *floating-point-pvar*.

In each processor, the value returned by **least-negative-float!!** is the negative floating point number closest to (but not equal to) zero that can be represented by the CM in the same floating-point format as *floating-point-pvar*.

float-epsilon!! *floating-point-pvar* [*Function*]

In each processor, the value returned by **float-epsilon!!** is the smallest positive floating-point number, *e*, that can be represented by the CM in the same floating point format as *floating-point-pvar* and for which

```
(not (= (float 1 e) (+ (float 1 e) e)))
```

is true when evaluated.

negative-float-epsilon!! *floating-point-pvar* [*Function*]

In each processor, the value returned by **negative-float-epsilon!!** is the smallest negative floating-point number *e* that can be represented by the CM in the same floating point format as *floating-point-pvar* and for which

```
(not (= (float 1 e) (- (float 1 e) e)))
```

is true when evaluated.

7.5 Logical Operations on Integer Pvars

lognand!! *integer-pvar1 integer-pvar2* [*Function*]
lognor!! *integer-pvar1 integer-pvar2* [*Function*]
logandc1!! *integer-pvar1 integer-pvar2* [*Function*]
logandc2!! *integer-pvar1 integer-pvar2* [*Function*]
logorc1!! *integer-pvar1 integer-pvar2* [*Function*]
logorc2!! *integer-pvar1 integer-pvar2* [*Function*]

These functions take two integer pvars and, within each processor, perform a bit-wise logical operation on the components of the argument pvars. The functions each return an integer pvar that contains the results of the logical operation. Like their Common Lisp analogs, these functions are not associative; they take exactly two arguments. Also like their Common Lisp analogs, these six functions are the nontrivial results of combining the basic logical operations AND, OR, and NOT. The equivalences are:

```
(lognand!!  n1 n2)   <=>   (lognot!! (logand!! n1 n2))
(lognor!!   n1 n2)   <=>   (lognot!! (logior!! n1 n2))
(logandc1!! n1 n2)   <=>   (logand!! (lognot!! n1) n2)
(logandc2!! n1 n2)   <=>   (logand!! n1 (lognot!! n2))
(logorc1!!  n1 n2)   <=>   (logior!! (lognot!! n1) n2)
(logorc2!!  n1 n2)   <=>   (logior!! n1 (lognot!! n2))
```

boole!! *op-pvar integer-pvar1 integer-pvar2* [*Function*]

The function **boole!!** permits the user to specify different logical operations be performed in different CM processors. It takes an operation pvar and two integer pvars; it returns an integer pvar that contains, in each processor, the result of the specified operation on the two integer components.

The following Common Lisp constants are acceptable as components of the *op-pvar* argument:

boole-clr	**boole-and**	**boole-1**	**boole-andc1**
boole-set	**boole-ior**	**boole-2**	**boole-andc2**
boole-eqv	**boole-nor**	**boole-c1**	**boole-orc1**
boole-xor	**boole-nand**	**boole-c2**	**boole-orc2**

For example,

```
(boole!! (!! boole-and) n1 n2) <=> (logand!! n1 n2)
```

Or, to have **boole-and** execute in all odd processors and **boole-ior** execute in all even processors, do:

```
(boole!! (if!! (oddp!! (self-address!!))
               (!! boole-and)
               (!! boole-ior))
         n1 n2)
```

logbitp!! *index-pvar integer-pvar* [*Function*]

This predicate function is true in each processor where the bit in *integer-pvar* whose index is *index-pvar* is a one-bit; otherwise it is false. The behavior is:

```
(logbitp!! k n) <=>
    (plusp!! (ldb!! (byte!! k (!! 1)) n))
```

logtest!! *integer-pvar1 integer-pvar2* [*Function*]

This predicate function is true in each processor where any of the one-bits in *integer-pvar1* is also a one-bit in *integer-pvar2.* The behavior is:

```
(logtest!! x y) ≡ (not!! (zerop!! (logand!! x y)))
```

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

This function determines, in each processor, the number of bits in that processor's component of *integer-pvar* and returns a non-negative integer pvar containing the results. If the component of *integer-pvar* is positive, then the one-bits in its binary representation are counted. If the component of *integer-pvar* is negative, then the zero-bits in its two's-complement binary representation are counted.

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

This function determines, in each processor, the number of bits required to represent that processor's component of *integer-pvar;* it returns a non-negative integer pvar containing the results.

A signed number requires (**1+!!** (**integer-length!!** *integer-pvar*)) bits to represent the integer in signed two's-complement form. For example,

```
(integer-length!! (!! 0))  <=> (!! 0)
(integer-length!! (!! 1))  <=> (!! 1)
(integer-length!! (!! 3))  <=> (!! 2)
(integer-length!! (!! 4))  <=> (!! 3)
(integer-length!! (!! 7))  <=> (!! 3)
(integer-length!! (!! -1)) <=> (!! 0)
(integer-length!! (!! -4)) <=> (!! 2)
(integer-length!! (!! -7)) <=> (!! 3)
(integer-length!! (!! -8)) <=> (!! 3)
```

7.6 Arithmetic Operations on Integer Pvars

The functions in this section are the parallel equivalents of the Common Lisp functions **gcd** and **lcm**.

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

This function takes zero or more integer pvars and computes, in each processor, the greatest common divisor of all of the argument pvar components in that processor. The function always returns a non-negative integer pvar. Specifically:

- If no arguments are given, 0 is returned in each processor.
- If one argument is given, its absolute value is returned in each processor.
- If two arguments are given, the **gcd** of the two pvar components is returned in each processor.
- If three or more arguments are given, the behavior is:

```
(gcd!! a b c ... z) <=> (gcd!! (gcd!! a b) c ... z)
```

lcm!! *integer-pvar* **&rest** *integer-pvars* [*Function*]

The function **lcm!!** takes one or more integer pvars and computes, in each processor, the least common multiple of the argument pvar components in that processor. It always returns a non-negative integer pvar. Specifically:

- If one argument is given, its absolute value is returned in each processor.

- If two arguments are given, the **lcm** of the two pvar components is returned in each processor.
- If three or more arguments are given, the behavior is:

```
(lcm!! a b c ... z) ≡ (lcm!! (lcm!! a b) c ... z)
```

- If one or more arguments (component values) are zero, then the result is zero.
- For two arguments that are not both zero,

```
(lcm!! a b) <=>
    (truncate!! (abs!! (*!! a b)) (gcd!! a b))
```

7.7 Byte Manipulation Function

*Lisp provides parallel equivalents of all Common Lisp byte manipulation functions. Here, as in Common Lisp, a byte is defined as an arbitrary number of contiguous bits. Also as in Common Lisp, many of these functions take an object called a *byte specifier* to designate a specific byte position within an integer. A *parallel byte specifier* is constructed by the function **byte!!**.

byte!! *size-pvar position-pvar* [*Function*]

This function is the parallel equivalent of the Common Lisp function **byte**. It takes two integer pvars representing the size and position of a byte pvar. For instance, a *size-pvar* of **(!! 16)** and a *position-pvar* of **(!! 3)** specify, in each processor, a 16-bit byte that starts at bit 3 (zero-based) of an integer pvar to be manipulated by one of the byte manipulation functions.

The arguments *size-pvar* and *position-pvar* may contain different values in each processor. The return value of **byte!!** is a byte specifier pvar suitable for use as an argument to byte-manipulation functions such as **ldb!!** and **dpb!!**.

byte-size!! *bytespec-pvar* [*Function*]

The function **byte-size!!** takes a byte specifier pvar—the result of a call to **byte!!**—and returns a copy of the originally specified *size-pvar* as an integer pvar. Thus:

```
(byte-size!! (byte!! size pos)) <=>  size
```

byte-position!! *bytespec-pvar* [*Function*]

The function **byte-position!!** takes a byte specifier pvar—the result of a call to **byte!!**—and returns an integer pvar that is a copy of the originally specified *position-pvar*. Thus:

```
(byte-position!! (byte!! size pos)) <=> pos
```

ldb!! *bytespec-pvar integer-pvar* [*Function*]

The function **ldb!!** is similar to the function **load-byte!!** and is the parallel equivalent of the Common Lisp function **ldb**. The *bytespec-pvar* specifies a byte of *integer-pvar* to be extracted. The result is returned as a non-negative integer pvar. The following forms are equivalent.

```
(load-byte!! integer-pvar position-pvar size-pvar)
<=>
(ldb!! (byte!! size-pvar position-pvar) integer-pvar)
```

ldb-test!! *bytespec-pvar integer-pvar* [*Function*]

This function is a predicate test and the parallel equivalent of **ldb-test**. It returns **t** in those processors where the byte field of *integer-pvar* specified by *bytespec-pvar* is non-zero. Elsewhere, it returns **nil**.

dpb!! *newbyte-pvar bytespec-pvar integer-pvar* [*Function*]

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

The function **dpb!!** returns an integer pvar that is the same as *integer-pvar* with the exception that the bits specified by *bytespec-pvar* receive their values from *newbyte-pvar*. The following forms are equivalent.

```
(deposit-byte!!
  integer-pvar position-pvar size-pvar newbyte-pvar)
<=>
```

```
(dpb!!
  newbyte-pvar (byte!! size-pvar position-pvar) integer-pvar)
```

mask-field!! *bytespec-pvar integer-pvar* [*Function*]

The function **mask-field!!** is the parallel equivalent of the Common Lisp function **mask-field**. It is similar to **ldb!!**; however, the result contains, for each processor, the byte of *integer-pvar* that is in the position specified by *bytespec-pvar*, rather than in position 0 as with **ldb!!**. The result therefore agrees with *integer-pvar* in the byte specified, but has zero-bits everywhere else.

The behavior is:

```
(mask-field (byte!! s p) n) <=>
    (logand!! n (dbp!! (!! -1) (byte!! s p) 0))
```

7.8 Conversions between Integers and Gray Code

The following two functions convert between integers and Gray code. Gray code is used in NEWS addressing in CM System Software supporting the Connection Machine model CM-2. See the *System Front Ends Release Notes*.

integer-from-gray-code!! *integer-pvar* [*Function*]

This function treats each component of the argument pvar as a Gray-coded integer and converts it to a non–Gray-coded integer. The *integer-pvar* argument should contain unsigned integers.The function returns a pvar containing the unsigned results. The binary reflected Gray code is used.

gray-code-from-integer!! *integer-pvar* [*Function*]

This function converts each integer component of the argument pvar into a Gray code representation. The *integer-pvar* argument should contain unsigned integers. The function returns a pvar containing the unsigned results.The binary reflected Gray code is used.

7.9 The Front-End Pvar Type

The front-end type is provided to allow *Lisp to manipulate front-end objects that cannot be represented on the CM.

(pvar front-end) [*Pvar Type*]

This is the type specifier for a pvar containing, in each active processor, a pointer to a front-end object. A single front-end object is pointed to by all active processors.

The use of front-end pvars is only sensible with certain types of *Lisp operations: those which access, move, or compare data, but do not combine or compute with it. The following list is representative of operations that may take front-end pvar arguments:

eq!!	**if!!**	**news!!**
pref!!	**pref**	***pset**
scan!! with copy!!	***set**	**setf of pref**

front-end!! *scalar-object* [*Function*]

This function returns a pvar of type **(pvar front-end)**. Note that a general pvar—that is, a pvar of type **(pvar t)**—can store a front-end pvar.

The argument *scalar-object* must be an object allocated on the front end computer.

front-end-p!! *pvar* [*Function*]

This function tests *pvar* and returns **t** in those processors containing pointers to a front-end object and **nil** elsewhere. Note that if *pvar* is a general pvar, **t** could be returned in some processors while **nil** is returned in others.

7.10 *Lisp Error Checking

*Lisp version 5.0 includes enhanced error checking facilities. There are two basic kinds of error checking performed by *Lisp: compile-time error checking and run-time error-checking. Both the compiler and the interpreter perform a certain amount of error checking automatically. For example, many *Lisp functions check for floating-point overflow and divide-by-zero errors. In addition, the *Lisp compiler and interpreter each have a *safety* level. The *Lisp compiler safety settings (determined by the

value of the variable ***safety***) are documented in the **Lisp Compiler Guide*. In what follows, the *Lisp interpreter safety is described.

interpreter-safety [*Variable*]

This variable determines the amount of run-time error checking performed by those *Lisp operations that are capable of it. The value of ***interpreter-safety*** must be an integer between 0 and 3, inclusive. The effect of each setting is given below.

0	most run-time error checking disabled
1	minimal run-time error checking; for any error signaled, an error message is not emitted until the next time a value is read from the CM
2	do not use this setting; reserved for future expansion
3	maximum run-time error checking; error messages emitted immediately

The value of ***interpreter-safety*** may be set in two ways. First, the function ***cold-boot** now takes an optional **:safety** keyword argument. If specified, the value of :safety sets both the *Lisp ***interpreter-safety*** variable and Paris safety. (For information on Paris safety, see the Paris documentation.) Thus,

```
(*cold-boot :dimensions '(2 36 36) :safety 3)
```

initializes the CM in a three-dimensional virtual processor configuration and sets both Paris safety and ***interpreter-safety*** to 3.

To set the value of ***interpreter-safety*** during a session, simply use the **setq** function:

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

This causes maximum run-time error checking to be performed.

Examples:

The interactions shown below demonstrate how ***interpreter-safety*** affects error messages. Assume we begin with an ***interpreter-safety*** value of 3, using Symbolics Common Lisp. (The screen display for Lucid Common Lisp is slightly different.)

```
*interpreter-safety* => 3
```

As safety level 3, errors are reported immediately. A division by zero is attempted.

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

/!!-2
   Arg 0 (A): #<FIELD-Pvar 4-2>
   Arg 1 (B): #<FIELD-Pvar 12-1>
s-A,  :Ignore Error.
s-B:   Display Processors With Error.
s-C:   Display Value in Processors with Error.
s-D:   Display Selected Processors.
s-E:   Display Value in Selected Processors.
s-F:   Display Value in All Processors.
s-G,  :Return to Lisp Top Level in Dynamic Lisp Listener 1
s-H:   Restart process Dynamic Lisp Listener 1

 Return to Lisp Top Level in Dynamic Lisp Listener 1
 Back to Lisp Top Level in Dynamic Lisp Listener 1.
```

Now ***interpreter-safety*** is set to 1.

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

At this setting, any error is reported only after a value has been read out of the CM. Another division by zero is attempted:

```
(/!! (!! 3) (!! 0))
#<FLOAT-Pvar 52-32>
;; Notice that no error has been signaled yet.
;; Now we read a value out of the CM.
(pref (!! 0) 0)
 Error: Error while accessing *UC-OUTPUT-FIFO-READ-ADDRESS*.
 The result of a (two argument) float /!! overflowed

CMI::WAIT-UNTIL-FEBI-OUTPUT-FIFO-NOT-EMPTY
s-A,  :Return to Lisp Top Level in Dynamic Lisp Listener 1
s-B:   Restart process Dynamic Lisp Listener 1

 Return to Lisp Top Level in Dynamic Lisp Listener 1
 Back to Lisp Top Level in Dynamic Lisp Listener 1.
```

Finally, ***interpreter-safety*** is set to 0.

```
(*warm-boot)
(setq *interpreter-safety* 0)
```

At this setting, no error is ever generated:

```
(/!! (!! 3) (!! 0))
#<FLOAT-Pvar 20-32>
(pref (!! 0) 0)
0
```

7.11 New Debugging Features

The *Lisp debugging tools have been updated with the release of Version 5.0. The **ppp** macro has been augmented and several related tools have been added.

ppp!! *pvar* **&rest** *keyword-args* [*Macro*]

The function **ppp!!** is identical to **ppp** except that it returns its pvar argument. The argument *pvar* may be any pvar. The *keyword-args* are identical to those for **ppp**. (See the **Lisp Reference Manual.*)

pppdbg *pvar* **&** **rest** *keyword-args* [*Macro*]

This macro is equivalent to **ppp**, except that the **:title** keyword argument defaults, not to **nil** (no title), but to the form that is evaluated to provide the pvar argument for **ppp**. The argument *pvar* may be any pvar. The *keyword-args* are identical to those for **ppp**. (See the **Lisp Reference Manual.*)

Examples:

```
(pppdbg (!! 2) :end 10)

(!! 2): 2 2 2 2 2 2 2 2 2 2

(pppdbg random-pvar :end 10)
```

```
random-pvar: 0 3 8 7 2 9 8 7 5 2
```

ppp-address-object *address-object-pvar* **&rest** *keyword-args* [*Macro*]

This prints out a pvar of type **(pvar address-object)** in a format that is easily understood. The geometry-id, cube-address, and each grid-address per processor are printed. See chapter 6 for information about address objects.

Only the **:start**, **:end**, **:title** and **:mode** keyword arguments are allowed. Otherwise the function is identical to **ppp**.

Example:

```
(ppp-address-object
  (grid!! (!! 1) (!! 2) (self-address!!)) :end 5)

Single cached geometry id: 25, Rank: 3
Cube Address      : 129 131 133 135 145
Grid Coordinate 0: 1 1 1 1 1
Grid Coordinate 1: 2 2 2 2 2
Grid Coordinate 2: 0 1 2 3 4
NIL
```

ppp *pvar* **&rest** *keyword-args* [*Macro*]

This macro is documented in the **Lisp Reference Manual*. However, the keyword **:ordering** is new with Version 5.0 and is described below.

The **:ordering** keyword argument to **ppp** and related operations takes a list of integers specifying axes. It is valid only when used in conjunction with the **:grid** value of the **:mode** keyword and most useful for printing a pvar defined in a VP set of more than two dimensions. With the **ppp :ordering** keyword, the user can specify how 'slices' of an *n*-dimensional area are to be displayed. The last two axes specified are the two axes which are shown in a single slice.

Example:

```
(def-vp-set 3d '(16 16 2))
=> 3D
(*with-vp-set 3d
 (ppp (self-address!!) :mode :grid :ordering '(0 1 2)
                       :end '(1 16 2)))

(0 1 2)

DIMENSION 0, COORDINATE 0

     DIMENSION 1    ----->

0 4 8 12 64 68 72 76 192 196 200 204 128 132 136 140
256 260 264 268 320 324 328 332 448 452 456 460 384 388 392 396

(*with-vp-set 3d
  (ppp (self-address!!) :mode :grid :ordering '(2 1 0)
                        :end '(16 16 1)))

(2 1 0)

DIMENSION 2, COORDINATE 0

     DIMENSION 1    ----->

0 4 8 12 64 68 72 76 192 196 200 204 128 132 136 140
1 5 9 13 65 69 73 77 193 197 201 205 129 133 137 141
2 6 10 14 66 70 74 78 194 198 202 206 130 134 138 142
3 7 11 15 67 71 75 79 195 199 203 207 131 135 139 143
16 20 24 28 80 84 88 92 208 212 216 220 144 148 152 156
17 21 25 29 81 85 89 93 209 213 217 221 145 149 153 157
18 22 26 30 82 86 90 94 210 214 218 222 146 150 154 158
19 23 27 31 83 87 91 95 211 215 219 223 147 151 155 159
48 52 56 60 112 116 120 124 240 244 248 252 176 180 184 188
49 53 57 61 113 117 121 125 241 245 249 253 177 181 185 189
50 54 58 62 114 118 122 126 242 246 250 254 178 182 186 190
51 55 59 63 115 119 123 127 243 247 251 255 179 183 187 191
32 36 40 44 96 100 104 108 224 228 232 236 160 164 168 172
33 37 41 45 97 101 105 109 225 229 233 237 161 165 169 173
34 38 42 46 98 102 106 110 226 230 234 238 162 166 170 174
35 39 43 47 99 103 107 111 227 231 235 239 163 167 171 175
```

Chapter 8

Parallel Variable Types

This chapter describes the different types of parallel variables, or *pvars*, available in *Lisp. The following new pvar types are introduced with *Lisp Version 5.0: character, string-char, array, front-end, complex, structure. For these, as well as for previously implemented pvar types, this chapter discusses the rules of type coercion and the semantics of using ***set** across various data types.

Version 5.0 introduces the pvar property of changing size without changing type. This is known as being *mutable*. Many pvar types may be declared mutable—with the advantage that declarations can be made without exact length specifications.

Previous versions of *Lisp implemented the indefinite pvar type, known as the *general pvar type* and declared as **(pvar t)** . A general pvar may now be declared mutable with the type specification **(pvar *)**. This type of pvar is called a *general mutable pvar.* Whereas in previous versions of *Lisp, pvars allocated without declarations defaulted to type **(pvar t)**, undeclared pvars now default to type **(pvar *)**. This can result in better performance for interpreted *Lisp code without pvar declarations. This chapter highlights the automatic type conversion that takes place when general mutable pvars are used.

8.1 Pvar Types

A pvar is defined by the kind of values that can be stored in it. The following pvars types are supported in *Lisp:

boolean	
unsigned-byte	**signed-byte**
defined-float	**complex**
string-char	**character**
array	**structure**
front-end	**general**

For most pvar types, *Lisp provides several equivalent forms that may be used in declarations. For instance, given any valid pvar type specifier **(pvar x)**, **x-pvar** is also a valid type specifier.

Each pvar type is listed below with equivalent type forms. Each pair of forms separated by <=> are equivalent and may be used interchangeably within ***proclaim**, **declare**, and **the** forms.

general

```
(pvar t) <=> general-pvar
```

front-end

```
(pvar front-end) <=> front-end-pvar
```

boolean

```
(pvar boolean) <=> boolean-pvar
```

signed-byte

```
(pvar (signed-byte width)) <=> (signed-pvar width)
```

unsigned-byte

```
(pvar (unsigned-byte width))  <=> (unsigned-byte-pvar width)
```

defined-float

```
(pvar (defined-float significand exponent))
float-pvar <=> (pvar (defined-float * * ))
```

```
short-float-pvar <=> (pvar short-float)
<=> (pvar (defined-float 15 8))

single-float-pvar <=> (pvar single-float)
<=> (pvar (defined-float 23 8))

double-float-pvar <=> (pvar double-float)
<=> (pvar (defined-float 52 11))

long-float-pvar <=> (pvar long-float)
<=> (pvar (defined-float 74 21))

(pvar extended-float) <=> (pvar (defined-float 96 31))
```

complex

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

complex-pvar <=> (pvar (complex (defined-float * *)))

short-complex-pvar <=> (pvar (complex short-float))
<=> (pvar (complex (defined-float 15 8)))

single-complex-pvar <=> (pvar (complex single-float))
<=> (pvar (complex (defined-float 23 8)))

double-complex-pvar <=> (pvar (complex double-float))
<=> (pvar (complex (defined-float 52 11)))

long-complex-pvar <=> (pvar (complex long-float))
<=> (pvar (complex (defined-float 74 21)))
```

character

```
(pvar character) <=> character-pvar
```

string-char

```
(pvar string-char) <=> string-char-pvar
```

array

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

structure

```
(pvar foo-struct) <=> foo-struct-pvar
```

where **foo-struct** is a parallel structure that has been defined with ***defstruct** as described in chapter 4

8.2 Mutable Pvars

If a pvar can change its type in any way, it is said to be *mutable*. Each pvar type form has zero or more parameters associated with it. To declare a pvar as mutable, specify the symbol * as the value of the parameter or parameters.

Examples:

```
(*let (mutable-signed-pvar)
  (declare (type (signed-pvar *) mutable-signed-pvar))
   ...)

(*proclaim '(type (pvar (defined-float * *)) mutable-float-pvar))
(*defvar mutable-float-pvar)
```

Each pvar type is governed by certain rules and restrictions concerning mutable type specification. These are detailed below in section 8.5, "Type Declaration and Coercion."

8.3 General Pvars

(pvar t) [*Type*]

A pvar that is declared explicitly as **(pvar t)** is a general pvar. Before a general pvar is initialized, it is referred to as void.

General pvars are allowed to contain, in different processors at the same time, data belonging to the following pvar types:

boolean	**character**
signed-byte	**defined-float**
complex	

Whenever a general pvar is used, *Lisp checks to see which data types it contains. Then, each data type the general pvar contains is checked to verify that it satisfies the domain requirements of the operation being performed. All this run-time checking takes time. General pvars therefore offer almost complete generality with a correspondingly severe reduction in run time efficiency.

When data of a particular type is stored in a general pvar, *Lisp ensures that the parameters for that type are identical across all the values of that type. If an attempt is

made to store pvars of the same type but with divergent parameters into a general pvar, *Lisp will coerce each pvar into a single type with identical parameters.

For example, when source values of type **(defined-float 52 8)** are stored in a general pvar containing values of type **(defined-float 23 11)**, the source values are copied and they and all the original values in the destination are coerced into type **(defined-float 52 11)**.

General pvars can receive data from any pvar that is not of type array or structure. When data of a particular pvar type is stored in a general pvar, *Lisp applies rules of type coercion specific to that pvar type. These rules are detailed in section 8.5 below.

8.4 Mutable General Pvars

Pvars that are not declared to be of a specific type default to a type known as *mutable general*. Before a mutable general pvar is initialized, it is said to be *void*. Notice that an uninitialized general pvar, **(pvar t)**, is also known as void.

(pvar *) [*Type*]

This is the form used within declarations to explicitly declare a mutable general pvar. For example, the forms

```
(*proclaim '(type (pvar *) random-mutable-pvar))
(*defvar random-mutable-pvar)
```

proclaim **random-mutable-pvar** to be a mutable general pvar and then allocate the pvar **random-mutable-pvar**.

Once a mutable general pvar has contained data of two or more distinct types, it loses its mutable quality and becomes a general pvar. For example, if a pvar declared to be of type **(pvar *)** has both integers and characters stored in it, it becomes a pvar of type **(pvar t)**.

If a mutable general pvar is void and a pvar of any single type is ***set** into it, then the mutable general pvar will assume the characteristics of that type, but the general mutable pvar will *not* lose its status as a general mutable pvar. For the purpose of this definition, the following pvar types are considered to belong to different types with respect to their effect on a mutable general pvar:

boolean	**signed-byte**
defined-float	**complex**
character	

The **signed-byte** pvar type is considered a super type that subsumes the **unsigned-byte** pvar type. Similarly, the **character** pvar type is considered to subsume the **string-char** pvar type. Thus, during a session, a mutable general pvar may hold both string-char and character data and still retain its status as a mutable general pvar. Similarly, if a mutable general pvar of type unsigned-byte has signed integers stored in it, it changes into a mutable general pvar of type signed-byte.

Given the above distinctions in type membership, as long as no data of a different type is ***set** into the mutable general pvar, the mutable general pvar will behave exactly as if it were a mutable pvar of the same type as the data stored it. This is significant because, if a mutable general pvar has held only one type of data, no tests are performed on the types it contains. Thus, the run-time execution speed of code using mutable general pvars that hold only one type of data is much faster than the execution speed of the same code using general pvars.

Mutable general pvars can receive data from a pvar that is not an array pvar or a structure pvar. Under one condition, an array pvar or a structure pvar *may* be stored in a mutable general pvar. If a mutable general pvar is void, it may be the destination pvar for a ***set** with array or structure source data. In this case, the mutable general pvar ceases to be a mutable general pvar and becomes an array or structure pvar of the same type and size as the source.

Within a ***set** form, a general pvar destination is always expanded as necessary to hold whatever size data is provided by the source. If the source is a general pvar, ***set** executes as though it were called once for each type of data contained in the source general pvar. Thus, given a source general pvar containing boolean, signed-byte, and complex data, the ***set** operation effectively performs the following sequence. First, only the processors containing boolean data are activated. Next, the boolean data is copied to a boolean pvar. Finally, ***set** is called with the general destination pvar and the boolean source pvar. This process is repeated for the signed-byte and complex data types.

If the source pvar of a ***set** with a destination of type general pvar is not a general pvar, ***set** works as described under each type of pvar, below.

8.5 Type Declaration and Coercion

Type declarations are useful for two reasons. First, interpreted code executes faster if type declarations are provided for all allocated pvars. Second, the *Lisp compiler will compile *Lisp code only if it employs pvars that are declared to be of a definite type. Neither general pvars nor general mutable pvars will compile.

This section defines the *Lisp rules of type declaration and coercion. For each *Lisp pvar type, the following questions are answered.

- Can data of this type be declared mutable?
- What happens when data of this type is stored in a general pvar?
- What types of data can be ***set** into a pvar of this type?

Note that, when source pvar values are ***set** into destination pvar locations, the source is first copied, then it is type converted if necessary. Finally the (possibly converted) copy of the source is stored in the destination.

(pvar boolean) [*Type*]
boolean-pvar [*Type*]

Boolean pvars have no parameters associated with them and are therefore never mutable.

When boolean data is stored in a general pvar, no type conversion is performed.

Within ***set** forms, boolean destination pvars can receive source data of type boolean only.

A general pvar can be ***set** into a boolean pvar if and only if all the active data in the general pvar is boolean.

(pvar front-end) [*Type*]

Front-end pvars have no parameters associated with them and are therefore never mutable.

When front-end data is stored in a general pvar, no type conversion is performed.

Within ***set** forms, front-end destination pvars can receive front-end source data only.

A general pvar can be ***set** into a front-end pvar if and only if all the active data in the general pvar is of type front-end.

(pvar string-char) [*Type*]
string-char-pvar [*Type*]

Pvars of type string-char have no parameters associated with them and therefore can never be declared as mutable.

When data of type string-char is put into a general pvar, it is first converted to type character.

Within ***set** forms, string-char destination pvars can receive source data of type string-char or of type character only. If the data, *source*, is of type character, then (***and (string-char-p!!** *source*)) must return **t**.

A general pvar can be ***set** into a string-char pvar if and only if all active data in the general pvar is of type string-char. That is, (***set** *destination source*) is valid if *destination* is a string-char pvar and if, for the general pvar *source*, (***and (string-char-p!!** *source*)) returns **t**.

(pvar character) [*Type*]
character-pvar [*Type*]

Character pvars have no parameters associated with them and therefore can never be declared as mutable.

When character data is put into a general pvar, no type conversion is performed.

Within ***set** forms, character destination pvars can receive source data of type string-char or of type character only.

A general pvar can be ***set** into a character pvar if and only if all the active data in the general pvar is of type string-char or of type character.

(pvar (unsigned-byte *length*)) [*Type*]
(field-pvar *length*) [*Type*]

Pvars of type unsigned-byte are also known as field pvars. They have one parameter associated with them, a length in bits. This length may be specified as any positive integer, or as *. Pvars declared as **(pvar (unsigned-byte *))** or **(field-pvar *)** are mutable. For instance,

```
(declare (type (field-pvar 16)) ubsixteen)
```

declares an unsigned-byte pvar of exactly 16 bits per processor. On the other hand,

```
(declare (type (field-pvar *)) ub-mut)
```

declares a mutable unsigned-byte pvar. Pvars declared as **(pvar (unsigned-byte *))** are initially allocated 1 bit per processor. They can, however, store unsigned values of any length.

When data of type unsigned-byte is put into a general pvar, it is first converted to an equivalent quantity of type signed-byte.

Within ***set** forms, destination pvars of type unsigned-byte can receive source data of type unsigned-byte or of type signed-byte only. If the source data is of type signed-byte, then all the data values must be non-negative; the source data is coerced to type unsigned-byte before storage is effected. If the destination is of type **(unsigned-byte *)**, then data of any number of bits is allowed. Otherwise, it must be possible to represent every active datum in the source using the number of bits specified for the destination's length.

A general pvar can be ***set** into a pvar of type unsigned-byte if and only if all the active data in the general pvar satisfies all the constraints detailed in the preceding paragraph.

(pvar (signed-byte *length*)) [*Type*]
(signed-pvar *length*) [*Type*]

Pvars of type signed-byte have one parameter associated with them, a length in bits. This length may be specified as any positive integer greater than 1, or as *. Pvars declared as **(pvar (signed-byte *))** are mutable. For instance,

```
(*proclaim '(type (pvar (signed-byte *)) s-mut))
```

proclaims a mutable signed-byte pvar. Signed-byte pvars declared with * are initially allocated 2 bits per processor. They can, however, store signed values of any length.

If source data of type signed-byte is moved into a general pvar, and if the source data length is larger than the length of the signed-byte data already contained in the destination, the signed-byte data already contained in the general pvar destination is sign-extended to accommodate the increased size.

Within ***set** forms, signed-byte pvars can receive source data of type unsigned-byte or of type signed-byte only. If the source data is of type unsigned-byte, it is coerced into type signed-byte before ***set** storage takes place. If the destination is of type **(signed-byte *)**, then source data any number of bits in length is allowed. Otherwise, it must be possible to represent every active datum in the source using the same number of bits as the destination.

A general pvar can be ***set** into a signed-byte pvar if and only if all the active data in the general pvar satisfies all the constraints detailed in the preceding paragraph.

(pvar (defined-float *significand exponent*)) [*Type*]

Pvars of type defined-float have two parameters associated with them: each defines the number of bits allocated per processor to store a portion of a floating-point number. The first parameter specifies the significand length; the second parameter specifies the exponent length.

The significand length may be any positive integer greater than or equal to 1 and less than **cm:*maximum-significand-length***. The exponent length may be any positive integer greater than or equal to 2 and less than **cm:*maximum-exponent-length***.

Mutable defined-float pvars are declared using * instead of a value for both significand length and exponent length. For example:

```
(declare (type (pvar (defined-float * *))) mut-float)
```

It is illegal to specify only one of these parameters as *. Mutable floating-point pvars are initially allocated 23 bits for the significand and 8 for the exponent, in each processor—with the sign bit, the total length is 32 bits..

When defined-float data is put into a general pvar, floating-point numbers with one representation may be coerced into floating-point numbers of another representation. If defined-float data with significand length *SL* and exponent length *EL* is copied into a general pvar containing defined-float data with significand length *GSL* and exponent length *GEL*, both the copied source and all floating-point values originally in the destination are coerced into a representation with **(max** *SL GLS***)** significand length and **(max** *GSL GEL*) exponent length. If there was originally no floating-point data in the general destination pvar, this has no effect; *GLS* and *GEL* are both zero in this case. If, however, floating-point data of a different representation resides in the destination pvar, such coercion may have repercussions with respect to overflow, underflow, precision, and accuracy.

The above rule of floating-point coercion for data stored in general pvars also applies to data stored in mutable defined-float pvars, pvars declared as **(pvar (defined-float * *)**.

Within ***set** forms, defined-float pvars can receive source data of type unsigned-byte, type signed-byte, or type defined-float only. If the source data is of type unsigned-byte or type signed-byte, a copy of it is converted to type defined-float using the *Lisp **float!!** operation. This implies that, even if the destination pvar is a mutable defined-float, it is an error to attempt to store unsigned-byte or signed-byte source data in that destination unless the source data can be represented in the same floating-point format as is the destination pvar data. If this error is made, an overflow error either is or is not be signaled, depending on safety level.

If the ***set** source data is of the same floating-point format as that of the destination, a simple data copy is done.

If the ***set** source data is of a floating-point format larger than the destination in either significand length or exponent length, and if the destination is not a mutable defined-float pvar, then it is an error.

If the ***set** destination is of type mutable defined-float, then both a copy of the source and the destination data are converted to a floating-point representation defined by the maximum of their significand and exponent lengths. After this conversion, a simple data copy is done.

A general pvar can be ***set** into a defined-float pvar if and only if all the active data in the general pvar satisfies the constraints in the preceding paragraphs.

(pvar (complex (defined-float *significand exponent*)**))** [*Type*]

*Lisp supports complex pvars with real and imaginary parts of type **defined-float**.

The restrictions on complex pvars parameters are identical to the restrictions on defined-float parameters. The real and imaginary parts are always of exactly the same type. Mutable complex pvars are declared with a * instead of with an integer value for each parameter. For example,

```
(*proclaim '(type (pvar (complex (defined-float * *))) cplx-mut)
```

declares a mutable complex pvar capable of storing variably sized complex numbers.

Since complex pvars can contain only defined-float components, the coercion rules for putting complex data into a general pvar are identical to those for defined-float quan-

tities. Note however that complex data is completely independent of defined-float data with respect to coercion: the existence of either type of data in a general pvar does not affect the representation of the other type.

The rule of complex coercion for data stored in general pvars also applies to data stored in mutable complex pvars.

Within ***set** forms, complex pvars can receive source data of type unsigned-byte, signed-byte, defined-float, or complex only. If the ***set** source data is of type unsigned-byte, signed-byte, or defined-float, it is coerced into the floating-point format determined by the complex destination, following the same rules as for defined-float. The source data is then converted to complex data of the same floating-point format as the destination, with 0.0 as its imaginary part. Finally, a simple data copy is done.

General pvars can be ***set** into complex pvars if and only if all the active data satisfies the constraints in the preceding paragraph.

(pvar (array *element-type dimensions*)) [*Type*]

Array pvars may not be declared mutable.

Array pvars may not be stored in general pvars. There is one exception: an array pvar *may* be stored in a void mutable general pvar. A void mutable general pvar is a pvar of type **(pvar *)** that has never had any data stored in it. When an array pvar is stored in a void mutable general pvar, that mutable general pvar becomes an array pvar with the same type and size as the array pvar which has been stored in it.

Within ***set** forms, array pvars can receive source data from other arrays pvars of the same shape. Effectively, ***set** is called on each element of the destination and source. The normal rules of type coercion with respect to the destination apply to ***set** operations acting on arrays.

(pvar *struct-name*) [*User-Defined Type*]

A pvar of type *struct-name* may be declared only after *struct-name* has been defined with ***defstruct**.

Structure pvars may not be declared mutable.

Structure pvars may not be stored in general pvars. There is one exception: a structure pvar *may* be stored in a void mutable general pvar. A void mutable general pvar is a pvar of type **(pvar *)** that has never had any data stored in it. When a structure pvar is

stored in a void mutable general pvar, that mutable general pvar becomes a structure pvar with the same type and size as the structure pvar which has been stored in it.

Within ***set** forms, structure pvars can receive source data from other structure pvars of exactly the same type. A simple bit copy is performed.

8.6 If No Processors Are Active, No Type Coercion Happens

It is an error for the source pvar in a ***set** form to be a void pvar when there are selected processors. If any processors are active, and if an attempt is made to use a void pvar as the source in a ***set** form, and if ***interpreter-safety*** is greater than 0, then *an error is signaled.* Conversely, if no processors are active it is not an error to use a void pvar as the source in a ***set** form.

The basic rule concerning ***set** when no processors are active is that nothing happens and nothing changes. No bits are copied and none of the the parameters of any pvar types are changed.

Examples:

```
(*let (x)
  ;; x is now (pvar *)
  (*when nil! (*set x (!! -3)))
  ;; x is still void. It is not of type signed-byte.
  )


(*all
  (*let (x)
    ;; x is void.
    (*set x (!! 3))
    ;; x is now of type unsigned-byte, and mutable to general
    (*when nil!! (*set x (!! 1.1)))
    ;; x is still of type unsigned-byte, mutable to general.
    ;; It is not of type (pvar t), which it would have been
    ;; had there been any processors selected.
    ))
```

```
(*let (x)
  (declare (type (pvar (unsigned-byte 8)) x))
  (*when nil!! (*set x (!! -1)))
  ;; This is not an error since nothing is proclaimed and
  ;; therefore no processor is receiving the improper negative
  ;; value.
  )
```

Experimental Features

A Warning About Experimental Features

Experimental features are features that are not completely stable. These features may be substantially changed in future *Lisp versions and Thinking Machines makes no guarantee that future implementations will be backwardly compatible with the current implementation. Users who wish to experiment with the capabilities permitted by these features do so at the risk of rewriting application code in the future to conform to a new, incompatible version.

Chapter 9

–>Experimental<–
Scanning with Segment Sets

The functionality of segmented scans has been enhanced. A new abstraction, termed a *segment set*, is now implemented. A segment is defined as a series of Connection Machine processors identified by contiguous send addresses. A segment set is defined as a collection of segments such that no processor is included in more than one segment. The segments included in a segment set need not span the entire address space.

Segment sets differ in two major characteristics from the segments optionally specified in a call to the **scan!!** function. First, segment sets are independent of the currently selected set (CSS). The segments specified as members of a segment set may include processors outside the CSS. Second, a segment set may be disjoint. That is, taken in ascending send address order, segments may be specified such that the first and last processors participating in any two segments need not be adjacent.

The use of segment sets for scanning is provided by two new *Lisp functions: **segment-set-scan!!** and **create-segment-set!!**.

9.1 Operations for Segmented Scans

A **segment-set-scan!!** operation works just like a **scan!!** operation except that it uses segment sets. It performs a specified associative binary *Lisp function over the values contained in the processors of each segment. This is done as a reduction analogous to the Common Lisp sequence function **reduce**. The cumulative result of the reduction is stored in each processor within a segment. For each segment, the scan operation is reinitiated; results obtained within one segment are not carried over into the next.

segment-set-scan!! *pvar scan-operator segment-set-pvar* [*Function*]
&key **:direction**
:check-for-processors-not-in-segment-set
:activate-all-processors-in-segment-set

The function **segment-set-scan!!** is similar to the function **scan!!**. It is used to perform scan operations over segment sets. At present, only scans using send address order are supported. (For a description of send addresses, see the definition of scan!! in section 6.2.) The **:include-self** option of **scan!!** is not used. Each processor always receives the result of applying the scan operation to all processors in its segment, including itself.

The return value of a call to **segment-set-scan!!** is a pvar containing scan results in each processor included in the specified segment set. In processors which have not participated in the scan operation, the result pvar contents are undefined.

The argument pvar may be any pvar acceptable to the function specified as the scan-operator argument.

The scan-operator may be one of the following associative binary parallel functions:

```
+!!, and!!, or!!, max!!, min!!, copy!!, logand!!, logior!!, logxor!!
```

The *segment-set-pvar* is a pvar returned by a call to the function **create-segment-set!!**. See the function description for **create-segment-set!!**, below.

The **:direction** keyword argument may be given as either **:forward** or **:backward** and defaults to **:forward**. A forward scan operation is performed in ascending send address order. Descending send address order is used if a backward direction is specified.

The **:check-for-processors-not-in-segment-set** keyword takes a boolean value and defaults to **nil**. If **t** is specified, **segment-set-scan!!** checks for processors which are in the CSS but which are not included in the segment set. If any are found, an error is signaled. If the default is used, the pvar value in processors which are in the CSS but which are not included in the segment set are simply ignored.

The **:activate-all-processors-in-segment-set** keyword takes a boolean value and defaults to **t**. If the default is used, all processors in the segment set are activated for the duration of the **segment-set-scan!!** operation. If **nil** is specified, the scan operation skips the pvar value in any processor that is not in the CSS, regardless of whether that processor is included in a segment of the segment set. This can fragment segments by allowing "holes" of deactivated processors. When a scan encounters a segment thus fragmented, it ignores any deactivated processors and carries the cumulative value of the scan into the next active processor in the segment.

Notice that the last option enables scans that operate only in those processors both active when the function is entered and inside one of the segments defined by the segment set.

create-segment-set!! *start-segment-pvar end-segment-pvar* [*Function*]

This function returns a segment set structure suitable for use as the third argument in a call to the **segment-set-scan!!** operation.

The two arguments to **create-segment-set!!** specify which processors are included in the segments of the segment set. These are boolean pvars, one or the other but not both of which may be **nil!!**.

The *start-segment-pvar* argument may be **nil!!** or it may contain **t** in each processor which starts a segment and **nil** in all other processors. The *end-segment-pvar* argument may be **nil!!** or it may contain **t** in each processor which ends a segment and **nil** in all other processors.

With these arguments, it is possible to specify a segment set from which certain processors are entirely excluded. However, if either argument to **create-segment-set!!** is **nil!!**, adjacent segments are defined.

When constructing *start-segment-pvar* and *end-segment-pvar*, take care to properly interleave the starting and ending processors for each segment. It is an error to specify overlapping segments.

From the segment start and end information, a structure pvar is constructed. The structure pvar created by a call to **create-segment-set!!** is defined as follows:

```
(*defstruct segment-set
  (start-bits nil :type boolean)
  (end-bits nil :type boolean)
  (processor-not-in-any-segment nil :type boolean)
  (start-address 0
    :type (signed-byte 32)
    :cm-type (pvar (signed-byte(1+ *current-send-address-length*))))
  (end-address 0
    :type (signed-byte 32)
    :cm-type (pvar (signed-byte(1+ *current-send-address-length*))))
  )
```

The **start-bits** and **end-bits** slot pvars are identical to the *start-segment-pvar* and *end-segment-pvar* arguments provided to **create-segment-set!!**. The **processor-not-in-**

any-segment slot pvar is t in each processor excluded from the segments in the set and **nil** elsewhere.

The send address of every first and last processor in each segment is calculated and stored with the **segment-set** structure in the **start-address** and **end-address** slot pvars. In each processor that is included in a segment, the **start-address** slot pvar contains the send address of the first processor in the segment and the **end-address** slot pvar contains the send address of the last processor in the segment. For processors excluded from all segments in the set, the **start-address** and **end-address** slot pvars each contain -1.

Chapter 10

–>Experimental<–
Parallel Vector Functions

*Lisp provides a set of experimental operations on numeric vector pvars. These perform standard operations, such as vector addition and dot product, on numeric vector pvars. For top performance, a set of specialized routines are provided for vector pvars of element type **single-float**.

All the numeric data types supported by *Lisp are valid element types for the vector pvars used in the parallel vector operations. These include unsigned-byte, signed-byte, float and complex.

The normal rules of *Lisp contagion are followed. For instance, the addition of a signed vector pvar with a float vector pvar results in a float vector pvar. The scaling of a float vector pvar by a complex scalar yields a complex result vector pvar.

The normal rules of *Lisp sizing are supported. For example, adding two vector pvars with element types **(unsigned-byte 8)** yields a result vector with element type **(unsigned-byte 9)**.

In general, it is an error if the vectors provided to any vector math function are not all of the same length.

***vset-components** *vector-pvar* **&rest** *component-pvars* [*Defun*]

If there is a single *component-pvar* argument, then every element of *vector-pvar* is ***set** to it. If there are as many *component-pvar* arguments as there are elements in *vector-pvar*, then the jth element of *vector-pvar* is ***set** to the jth *component-pvar* argument. An error will be signaled if the number of *component-pvar* arguments is not either 1 or the number of elements in the *vector-pvar*.

v+!! *vector-pvar &rest more-vector-pvars* [*Function*]
v-!! *vector-pvar &rest more-vector-pvars* [*Function*]
v*!! *vector-pvar &rest more-vector-pvars* [*Function*]

These functions allocate a result vector pvar according to the standard *Lisp contagion and sizing rules, discussed above. The obvious operation is performed elementwise. If a single argument is given to v+!! or v*!! it is returned unchanged.

dot-product!! *vector-pvar1 vector-pvar2* [*Function*]

This function returns a scalar pvar of the proper type and size. In each processor, the inner product of the two vectors is returned. Thus,

```
(dot-product!! c1-pvar c2-pvar)
<=>
(reduce #'+ (map 'vector #'* c1 c2))
```

in every active processor.

cross-product!! *vector-pvar1 vector-pvar2* [*Function*]

In each processor, the cross product of the two vector pvars is computed. The result is returned as a vector pvar.

vabs-squared!! *vector-pvar* [*Function*]

This function returns a scalar pvar of the same type as *vector-pvar*—but, if the element type is unsigned or signed, of larger size.

Calling **(vabs-squared!!** *vector-pvar*) is equivalent to

```
(dot-product!! vector-pvar vector-pvar)
```

vabs!! *vector-pvar* [*Function*]

This function returns a scalar pvar of type float if the element type of *vector-pvar* is non-complex. If the element type of *vector-pvar* is complex, **vabs!!** returns a complex pvar. This function is equivalent to

```
(sqrt!! (vabs-squared!! vector-pvar))
```

vscale!! *vector-pvar scalar-pvar* [*Function*]

This function returns a vector pvar of the proper type and size according to the *Lisp contagion and sizing rules.

In each processor, each element of the input pvar, *vector-pvar*, is multiplied by the single element of *scalar-pvar* in that processor.

vscale-to-unit-vector!! *vector-pvar* [*Function*]

This function is equivalent to

```
(vscale!! vector-pvar (/!! (vabs!! vector-pvar)))
```

except that *vector-pvar* is evaluated once.

It is an error if **(vabs!!** *vector-pvar***)** is zero in any processor.

10.1 Experimental Special-Purpose Single-Float Vector Operations

The *Lisp Compiler does not currently *compile parallel vector math operations. Therefore, specialized functions are offered to optimize interpreted code.

Special versions of the above functions (plus a few more) are provided for vector pvars with element type **single-float**. The function names for these functions begin with the prefix "sf-", for single-float.

Also provided are even more specialized functions that take a vector pvar destination argument as well as the usual arguments. These functions avoid the execution time and memory utilization overhead of allocating temporary result vector pvars. Avoiding this overhead is especially important when operating on long vectors. The function names for these functions begin with the prefix "d", for destination.

All functions prefixed with "sf-" and "d" require vector pvars whose element type is **single-float**. Any non-vector pvar arguments must be of type **(pvar single-float)**. It is an error if these conditions are not observed.

Be aware that these functions perform no overflow, underflow, or divide by zero error checking. In other words, their behavior is like that of code compiled with ***safety*** set to 0. These functions are provided solely for speed.

***sf-vset-components** *vector-pvar* **&rest** *component-pvar* [*Defun*]

This operation is analogous to ***vset-components**.

sf-v+-constant!! *vector-pvar scalar-pvar* [*Function*]
sf-v--constant!! *vector-pvar scalar-pvar* [*Function*]
sf-v*-constant!! *vector-pvar scalar-pvar* [*Function*]
sf-v/-constant!! *vector-pvar scalar-pvar* [*Function*]

In each processor, these functions add, subtract, multiply, or divide each element of the input pvar, *vector-pvar*, by the input pvar, *scalar-pvar*, and return a pvar vector of element type **single-float**.

dsf-v+-constant!! *vdest vector-pvar scalar-pvar* [*Defun*]
dsf-v--constant!! *vdest vector-pvar scalar-pvar* [*Defun*]
dsf-v*-constant!! *vdest vector-pvar scalar-pvar* [*Defun*]
dsf-v/-constant!! *vdest vector-pvar scalar-pvar* [*Defun*]

These operations perform the same computation as their analogues above and store the result in *vdest*. The argument *vdest* must be a pvar vector of element type **single-float**.

sf-v+!! *vector-pvar* **&rest** *more-vector-pvars* [*Function*]
sf-v-!! *vector-pvar* **&rest** *more-vector-pvars* [*Function*]
sf-v*!! *vector-pvar* **&rest** *more-vector-pvars* [*Function*]

These functions are the single-float analogues of **v+!!**, **v-!!** and **v*!!**, respectively.

dsf-v+!! *vdest vector-pvar* **&rest** *more-vector-pvars* [*Defun*]
dsf-v-!! *vdest vector-pvar* **&rest** *more-vector-pvars* [*Defun*]
dsf-v*!! *vdest vector-pvar* **&rest** *more-vector-pvars* [*Defun*]

These operations perform the same computation as their analogues above and store the result in *vdest*. The argument *vdest* must be a vector pvar of element type **single-float**.

sf-vabs!! *vector-pvar* [*Function*]
sf-vabs-squared!! *vector-pvar* [*Function*]
sf-dot-product!! *vector-pvar1 vector-pvar2* [*Function*]
sf-vscale!! *vector-pvar scalar-pvar* [*Function*]
sf-vscale-to-unit-vector!! *vector-pvar* [*Function*]

These functions are the single float analogues of **vabs!!**, **vabs-squared!!**, **dot-product!!**, **vscale!!** and **vscale-to-unit-vector!!**, respectively.

sf-cross-product!! *vector-pvar1 vector-pvar2* [*Function*]

The arguments *vector-pvar1* and *vector-pvar2* must each be a pvar vector of type **(pvar (array single-float (3)))**.

In each processor, the cross product of the two vectors is computed and returned as a pvar vector of type **(pvar (array single-float (3)))**.

sf-vector-normal!! *vector-pvar1 vector-pvar2* [*Function*]

This function is equivalent to

```
(sf-vscale-to-unit-vector!!
   (sf-cross-product!! vector-pvar1 vector-pvar2))
```

dsf-vscale!! *vdest vector-pvar scalar-pvar* [**Defun*]
dsf-vscale-to-unit-vector!! *vdest vector-pvar* [**Defun*]
dsf-cross-product!! *vdest vector-pvar1 vector-pvar2* [**Defun*]
dsf-vector-normal!! *vdest vector-pvar1 vector-pvar2* [**Defun*]

These operations each perform the expected operation and store the result in *vdest*. The argument *vdest* must be a pvar vector of element type **(pvar single-float)**. For **dsf-cross-product!!** and **dsf-vector-normal!!**, the arguments *vector-pvar1* and *vector-pvar2* must each be a pvar vector of type **(pvar (array single-float (3)))**.

10.2 Serial Equivalents of the Single-Float Vector Operations

For symmetry, serial equivalents to the above functions are provided.

All these operations operate on any kind of numeric vector and perform according to the rules of Common Lisp contagion and coercion.

v+-constant *vector scalar* [*Function*]
v--constant *vector scalar* [*Function*]
v*-constant *vector scalar* [*Function*]
v/-constant *vector scalar* [*Function*]

v+ *vector* **&rest** *more-vectors* [*Function*]
v- *vector* **&rest** *more-vectors* [*Function*]
v* *vector* **&rest** *more-vectors* [*Function*]

dot-product *vector1 vector2* [*Function*]
vabs-squared *vector* [*Function*]
vabs *vector* [*Function*]
cross-product *vector1 vector2* [*Function*]
vscale *vector scalar* [*Function*]
vscale-to-unit-vector *vector* [*Function*]
vector-normal *vector1 vector2* [*Function*]
vfloor *vector* [*Function*]
vceiling *vector* [*Function*]
vround *vector* [*Function*]
vtruncate *vector* [*Function*]

Chapter 11

–>Experimental<–
Parallel Sequence Operations

*Lisp sequence operations are the parallel equivalents of Common Lisp sequence operations, with the exception of the following general restrictions and of the further restrictions stated in the operation definitions given in this chapter.

In *Lisp, the term *sequence pvar* refers to any one-dimensional array pvar, also called a *vector pvar*. This differs from Common Lisp, which defines the sequence data type to include both lists and vectors. *Lisp does not implement list pvars.

Array pvars in *Lisp are restricted to be of the same fixed size in each processor. The manner in which this restricts the generality of certain sequence functions is noted in the definitions that follow.

*Lisp provides the following subset of Common Lisp sequence functions, implemented to operate on sequence pvars.

subseq!!	**some!!**
copy-seq!!	**every!!**
length!!	**notany!!**
***nreverse**	**notevery!!**
reverse!!	**reduce!!**
***fill**	**find!!**
substitute!!	**position!!**
nsubstitute!!	**count!!**

As in Common Lisp, *Lisp includes variations on many of these operations. Such variations are indicated by the addition of "-if" or "-if-not" as a suffix .

11.1 Argument Conventions in Sequence Operations

The operation definitions that follow all use arguments denoted by the metavariable *sequence-pvar*. This argument must be an array pvar of one dimension.

Many definitions in this chapter include keyword arguments denoted by **:start** and **:end**. In all cases, these are used to define a subsequence pvar which is then acted upon by the operation. This use of **:start** and **:end** keyword parameters is consistent with Common Lisp.

Arguments to **:start** and **:end** must be positive integer pvars containing values within the range of 0 through $n + 1$, where n is the highest index of the sequence pvar and $n + 1$ is the length of the sequence pvar. The subsequence pvar thus defined includes elements indexed by the *start* value in each processor up to but excluding elements indexed by the *end* value. In any active processor, the start must be less than or equal to the end. The value of *start* defaults to **(!! 0)**. The value of *end* defaults to **(length!!** *sequence-pvar*). If different start and end values are stored in different processors, it is as though the processors contain subsequence vectors of different lengths.

Many sequence operations that require comparing sequence pvar elements with other given values will allow specification of a test other than the default, **eql!!**. An optional predicate function can be supplied as a keyword argument to either **:test** or **:test-not**.

Some sequence operations perform tests on sequence pvar elements. Where these operations require that part of an element be extracted and supplied to the test, a user-defined function may be provided as an argument to the keyword **:key**. Such key functions must take one argument: a sequence pvar element.

11.2 Simple Operations on Sequence Pvars

subseq!! *sequence-pvar start* **&optional** *end* [*Function*]

This function returns, in each processor, a sequence pvar of the same type as *sequence-pvar* and of length **(-!!** *end start*). The resulting sequence pvar contains a copy of the values of the elements found in *sequence-pvar*.

The argument *sequence-pvar* must be a sequence pvar. The arguments *start* and *end* must be non-negative integer pvars within the range of indices for *sequence-pvar*. Unlike most of the other sequence pvar operations, both *start* and *end* must contain uni-

form values in all active processors. Thus, the value of (**-!!** *end start*) must be the same across all active processors.

Example:

```
(setq abcd (typed-vector!! '(pvar character)
   (!! #\A) (!! #\B) (!! #\C) (!! #\D)))

(setq bc (subseq!! abcd (!! 1) (!!2))

(ppp (aref!! bc (!! 0) (!! 1)) :end 3)

=>#\B #\C #\B #\C #\B #\C
```

copy-seq!! *sequence-pvar* [*Function*]

This function returns a copy of *sequence-pvar*. The argument *sequence-pvar* must be a vector pvar.

length!! *sequence-pvar* [*Function*]

This function returns a positive integer pvar indicating the number of elements in *sequence-pvar*.

The argument *sequence-pvar* must be a vector pvar. The pvar returned by **length!!** holds the same value in each processor. Therefore:

```
(length!! sequence-pvar)
<=>
(!! (*array-total-size sequence-pvar))
```

***nreverse** *sequence-pvar* [**Defun*]

The function ***nreverse** destructively modifies *sequence-pvar* to contain its elements in reverse order. The argument *sequence-pvar* must be a vector pvar. For example:

```
(*nreverse (!! #(1 2 3 4))) => (!! #(4 3 2 1)).
```

reverse!! *sequence-pvar* [*Function*]

This function returns a sequence pvar that is a reversed copy of *sequence-pvar*. The argument *sequence-pvar* must be a vector pvar. The following equivalence always holds:

```
(reverse!! sequence-pvar)
<=>
(*nreverse (copy-seq!! sequence-pvar))
```

11.3 Mapping Predicates Over Sequence Pvars

some!! *predicate sequence-pvar* **&rest** *more-sequence-pvars* [*Function*]
every!! *predicate sequence-pvar* **&rest** *more-sequence-pvars* [*Function*]
notany!! *predicate sequence-pvar* **&rest** *more-sequence-pvars* [*Function*]
notevery!! *predicate sequence-pvar* **&rest** *more-sequence-pvars* [*Function*]

Each of these functions returns a boolean pvar indicating whether *predicate* is true for some, every, none, or not all sequence pvars given.

The function **some!!** returns **t** in each processor where *predicate* proves true for any element of the sequence in that processor. If *predicate* proves false for all elements of the sequence in a processor, **some!!** returns **nil** in that processor.

The function **every!!** returns **nil** in each processor where *predicate* proves false for any element of the sequence in that processor. If *predicate* proves true for all elements of the sequence in a processor, **every!!** returns **nil** in that processor.

The function **notany!!** returns **nil** in each processor where *predicate* proves false for all elements of the sequence in that processor. If *predicate* proves true for any element of the sequence in a processor, **notany!!** returns **nil** in that processor.

The function **notevery!!** returns **t** in each processor where *predicate* proves false for any element of the sequence in that processor. If *predicate* proves true for all elements of the sequence in a processor, **notany!!** returns **nil** in that processor.

The argument *sequence-pvar* must be a vector pvar, as must each &rest argument. All sequence pvars provided must be the same size. The argument *predicate* must be a test function that takes as many arguments as there are sequence pvars provided.

In each processor, the argument *predicate* is applied sequentially to each element, beginning with the 0th element and continuing until the termination criterion is met or all sequence pvars are exhausted.

reduce!! *function sequence-pvar* **&key :from-end** [*Function*]
:start :end
:initial-value

The function **reduce!!** operates in each processor to combine all the elements of *sequence-pvar*, two at a time, using *function*. A pvar containing the reduction result in each processor is returned. For example:

```
(reduce!! #'+!! number-sequence-pvar)
```

adds up all the elements of **number-sequence-pvar** in each processor.

The argument *function* must be a binary operation that accepts pvar arguments of the type contained in *sequence-pvar*. The argument *sequence-pvar* must be a vector pvar.

The keyword **:from-end** takes a boolean and defaults to **nil**. Reduction is left-associative in any processor with a **:from-end** value of **nil**. Otherwise, reduction is right-associative.

The keywords **:start** and **:end** take values as described at the beginning of this chapter and thereby define a subsequence of *sequence-pvar*. If a subsequence is defined by **:start** and **:end**, it is reduced using *function*.

The keyword **:initial-value** takes a pvar of the same type as the elements of *sequence-pvar*. If an **:initial-value** value is supplied, it is logically placed at the beginning of *sequence-pvar* or of the subsequence pvar and included in the reduction calculation. If **:from-end** is **t**, the value of **:initial-value** is logically placed at the end of *sequence-pvar* or of the subsequence pvar and included in the reduction calculation.

11.4 Operations Modifying Sequence Pvar

***fill** *sequence-pvar item* **&key :start :end** [**Defun*]

This function destructively modifies *sequence-pvar* or a subsequence of *sequence-pvar* by filling each element with *item*.

The argument *sequence-pvar* must be a vector pvar. The argument *item* must be a pvar of the same type as the elements of *sequence-pvar*.

The keywords **:start** and **:end** take values as described at the beginning of this chapter and thereby define a subsequence of *sequence-pvar*.

substitute!! *newitem olditem sequence-pvar* [*Function*]
&key :from-end :test :test-not
:start :end :count :key

substitute-if!! *newitem test sequence-pvar* [*Function*]
&key :from-end :start :end
:count :key

substitute-if-not!! *newitem test sequence-pvar* [*Function*]
&key :from-end :start :end
:count :key

The function **substitute!!** compares each element of *sequence-pvar* on a per-processor basis with *olditem*. The return value is a modified copy of *sequence-pvar* in which *newitem* has been substituted for each element that is the same as *olditem*.

The function **substitute-if!!** applies *test* to each element of *sequence-pvar*. Where *test* succeeds, **substitute!!** substitutes the local value of *newitem*. The return value is a modified copy of *sequence-pvar*.

The function **substitute-if-not!!** applies *test* to each element of *sequence-pvar*. Where *test* fails, **substitute-if-not!!** substitutes the local value of *newitem*. The return value is a modified copy of *sequence-pvar*.

The argument *sequence-pvar* must be a vector pvar. The arguments *newitem* and *olditem* must be pvars of the same type as the element type of *sequence-pvar*. These arguments may contain different values in different processors.

The argument *test* must be a scalar predicate in one argument.

The keyword **:from-end** takes a boolean pvar that specifies from which end of *sequence-pvar* in each processor the operation will take place. The default is **nil!!**, indicating a forward direction for all processors. A non-nil **:from-end** argument indicates the reverse direction but makes no difference unless a **:count** argument is also supplied.

The keywords **:test** and **:test-not** may not be used together. Arguments to these must be binary equivalence-testing functions. The comparison between *olditem* and *newitem* is made using **eql!!** unless another test function is supplied with either the **:test** or the **:test-not** keyword.

Arguments to the keywords **:start** and **:end** define a subsequence pvar to be operated on by **substitute!!** as described at the beginning of this chapter.

The **:count** keyword argument must be a positive integer pvar with values less than or equal to **(length!!** *sequence-pvar*). The function returns after *count* elements have satisfied the test.

The **:key** keyword accepts a user-defined function used to extract a part of an element of *sequence-pvar*. This key function must take one argument: an element of *sequence-pvar*.

nsubstitute!! *newitem olditem sequence-pvar* [*Function*]
&key :from-end :test :test-not
:start :end :count :key

nsubstitute-if!! *newitem test sequence-pvar* [*Function*]
&key :from-end :start :end
:count :key

nsubstitute-if-not!! *newitem test sequence-pvar* [*Function*]
&key :from-end :start :end
:count :key

These functions are the parallel equivalents of the Common Lisp **nsubstitute** functions.

The **nsubstitute!!** functions are destructive versions of the **substitute!!** functions. Upon return from an **nsubstitute!!** operation, *sequence-pvar* is modified.

11.5 Operations Searching Sequence Pvars

find!! *item sequence-pvar* [*Function*]
&key :from-end :test :test-not
:start :end :key :return-value-if-not-found

find-if!! *test sequence-pvar* [*Function*]
&key :from-end :start :end
:key :return-value-if-not-found

find-if-not!! *test sequence-pvar* [*Function*]
&key :from-end :start :end
:key :return-value-if-not-found

These functions are the parallel equivalents of the Common Lisp **find** functions except that an additional keyword, **:return-value-if-not-found**, is provided.

The function **find!!** searches *sequence-pvar* for elements that match *item*. It returns a pvar containing a copy of the first instance found in each processor. In any processor failing the search, the result contains *return-value-if-not-found*,if it was supplied and **nil** otherwise. Elements of *sequence-pvar* are tested against *item* with **eql!!** unless either *test* or *test-not* is supplied as an alternative.

The functions **find-if!!** and **find-if-not!!** are similar. They search *sequence-pvar* for elements that either do or do not pass *test* and return a copy of the first element found in each processor.

The argument *item* must be a pvar of the same type as the element type of *sequence-pvar*. The argument *sequence-pvar* must be a vector pvar of any *Lisp type.

The argument *test* must be a scalar predicate taking one argument of the same type as that of each element of *sequence-pvar*.

The keyword **:from-end** takes a boolean pvar that specifies from which end of *sequence-pvar* in each processor the operation will take place. The default is **nil!!**, indicating a forward direction for all processors.

The keywords **:test** and **:test-not** may not be used together. Arguments to these must be binary equivalence-testing functions. The comparison between *sequence-pvar* elements and *item* is made using **eql!!** unless either *test* or *test-not* is supplied.

Arguments to the keywords **:start** and **:end** define a subsequence pvar to be operated on, as described at the beginning of this chapter.

The **:key** keyword accepts a user-defined function used to extract a part of an element of *sequence-pvar*. This key function must take one argument: an element of *sequence-pvar*.

The keyword argument to **:return-value-if-not-found** must be a pvar and defaults to **nil!!**. The value of this pvar is returned in any processor where the search is not successful.

position!! *item sequence-pvar* [*Function*]
&key :from-end :test :test-not
:start :end :key

position-if!! *test sequence-pvar* [*Function*]
&key :from-end :start :end :key

position-if-not!! *test sequence-pvar* [*Function*]
&key :from-end :start :end :key

The **position!!** functions are very similar to the **find!!** functions. Here, however, it is not the found sequence pvar elements but their indices that are returned.

These functions are almost parallel equivalents of the Common Lisp **position** functions with two exceptions. Rather than returning **nil**, these return -1 in processors where the search fails. Each **position!!** function returns a 32 bit signed-byte pvar.

The function **position!!** searches *sequence-pvar* for elements that match *item*. It returns a pvar containing the index of the first match found in each processor. In any processor failing the search, the result contains -1. Elements of *sequence-pvar* are tested against item with **eql!!** unless either *test* or *test-not* is supplied as an alternative.

The functions **position-if!!** and **position-if-not!!** are similar. They search *sequence-pvar* for elements that either do or do not pass *test*.

The argument *item* must be a pvar of the same type as the element type of *sequence-pvar*. The argument *sequence-pvar* must be a vector pvar of any *Lisp type.

The argument *test* must be a scalar predicate taking one argument: a sequence element.

The argument *test* must be a scalar predicate taking one argument: a *sequence-pvar* element.

The keyword **:from-end** takes a boolean pvar that specifies from which end of *sequence-pvar* in each processor the operation will take place. The default is **nil!!**, indicating a forward direction for all processors.

The keywords **:test** and **:test-not** may not be used together. Arguments to these must be binary equivalence-testing functions. The comparison between *sequence-pvar* elements and *item* is made using **eql!!** unless either *test* or *test-not* is supplied.

Arguments to the keywords **:start** and **:end** define a subsequence pvar to be operated on, as described at the beginning of this chapter.

The **:key** keyword accepts a user-defined function used to extract a subsequence of *sequence-pvar*. This key function must take one argument: an element of *sequence-pvar*.

The keyword **:from-end** takes a boolean pvar that specifies from which end of *sequence-pvar* in each processor the operation will take place. The default is **nil!!**, indicating a forward direction for all processors.

The keywords **:test** and **:test-not** may not be used together. Arguments to these must be binary equivalence-testing functions. The comparison between *sequence-pvar* elements and *item* is made using **eql!!** unless either *test* or *test-not* is supplied.

Arguments to the keywords **:start** and **:end** define a subsequence pvar to be operated on, as described at the beginning of this chapter.

The **:key** keyword accepts a user-defined function used to extract a subsequence of *sequence-pvar*. This key function must take one argument: an element of *sequence-pvar*.

count!! *item sequence-pvar* [*Function*]
&key :from-end :test :test-not
:start :end :key

count-if!! *test sequence-pvar* [*Function*]
&key :from-end :start :end :key

count-if-not!! *test sequence-pvar* [*Function*]
&key :from-end :start :end :key

The **count!!** functions are very similar to the **find!!** functions. Here, however, the search continues until *sequence-pvar* is exhausted. A count of the *sequence-pvar* elements satisfying the search is returned. Each **count!!** function returns a 32 bit unsigned-byte pvar.

The function **count!!** searches *sequence-pvar* for elements that match *item*. It returns a pvar containing a count of the matching elements found in each processor. Elements of *sequence-pvar* are tested against *item* with **eql!!** unless either *test* or *test-not* is supplied as an alternative.

The functions **count-if!!** and **count-if-not!!** are similar. They search *sequence-pvar* for elements that either do or do not pass *test* and return a count.

The argument *item* must be a pvar of the same type as the element type of *sequence-pvar*. The argument *sequence-pvar* must be a vector pvar of any *Lisp type.

Appendixes

Appendix A

The Relationship between the CM-2 Architecture, Paris, and *Lisp

This appendix explains which advanced features of the CM-2 architecture are accessible from *Lisp, Version 5.0 and which are not. The advanced CM-2 features considered here are:

- Sprint routing
- Backward routing
- Combined routing
- Indirect addressing
- Floating-point accelerator
- Spreads

In each case, the *Lisp features provided to take advantage of the CM-2 hardware architecture are briefly described. For descriptions of the Paris software features mentioned here, see the *Paris Reference Manual*, Version 5.0.

A.1 Sprint Routing

Before the release of the Connection Machine System Software Version 5.0, routing using a virtual processor (VP) ratio of n took time that was not proportional to n, but rather to some higher power of n. Paris sprint routing software now allows routing using a VP ratio of n to take time proportional to n.

The choice between regular routing and sprint routing is made by the Paris routing software based on the current VP ratio. The *Lisp user has no control over this decision. Paris chooses to do normal routing when a low VP ratio is in effect and sprint routing when a high VP ratio is in effect. For example, with a VP ratio of 1, Paris will effect normal routing. In contrast, with a VP ratio of 16, Paris will chose sprint routing.

A.2 Backward Routing

Backward routing allows a *Lisp user to execute a **pref!!** with collisions in nearly the same time as a **pref!!** with no collisions, albeit with a significant tradeoff in space. There is currently no way to predict how much memory will be used. The amount of memory used is a function of the VP ratio and the exact routing pattern used. It is roughly on the order of 100 times the VP ratio in bits.

In order to cause the Connection Machine to do backward routing, the semantics of **pref!!** have been changed slightly for Release 5.0. The default value of the **:collision-mode** keyword has been changed from **:collisions-allowed** to **nil**. If no value is specified for this keyword, or if the value specified is **nil**, then backward routing will be used. If any of the previously defined keywords are specified (**:no-collisions**, **:collisions-allowed**, or **:many-collisions**), then backward routing will *not* be used.

A detailed description of **pref!!** can be found in the *Supplement to the *Lisp Reference Manual* chapter 6, entitled "*N*-Dimensional Interprocessor Communication."

A.3 Combining Routing

Combining routing has the effect of making ***pset** calls that use combiners such as **:add** and **:logior** operate much faster than on a CM1. Previously, ***pset** combining was done exclusively at the destination processors in a serial fashion. Now, the combining is done inside the router as the messages are traversing the hypercube.

This feature is used automatically by the Paris send instructions whenever possible. The *Lisp user has no control over whether the combining router is used or not.

A.4 Indirect Addressing

In *Lisp the term *indirect addressing* is used to refer to pvar array referencing that uses different index values in different processors. This type of array referencing is generally slower than array referencing that uses index pvars containing the same values in each processor. The CM-2 is capable of doing fast indirect addressing when data is represented *sideways*. (See the function definitions for **TRANSPOSE-DATA**, **AREF-32-2L**, and **ASET-32-2L** in the *Paris Reference Manual*, Version 5.0.)

To use this capability, two new experimental *Lisp functions, ***sideways-array** and **sideways-aref!!** are introduced with Release 5.0. The function ***sideways-array** takes an existing array pvar and forces it to be represented *sideways*. After this is done, **sideways-aref!!** and **(*setf (sideways-aref!! ...))** can be used to read non-uniformly positioned array pvar elements out of each processor. The use indirect addressing in *Lisp is described in detail in chapter 3 of the *Supplement to the *Lisp Reference Manual.*

A.5 Floating-Point Accelerator

The CM-2 floating-point accelerator operates on data turned sideways. Paris floating-point instructions must therefore turn data sideways *before* it can be processed by the floating-point hardware and invert it before writing it back to memory. At low VP ratios this can be inefficient. Therefore, keeping floating-point data sideways can result in performance improvements. This CM-2 capability is, however, not yet supported in software.

When a CM-2 has floating-point hardware, Paris and *Lisp use it by turning data sideways before executing any floating-point operation and returning it to a normal representation afterwards. From *Lisp, there is currently no way to have floating-point data that is turned sideways operated on by the *Lisp floating-point functions.

PERFORMANCE NOTE

On a CM-2 with the special floating-point accelerator, *Lisp code that uses float pvars of type **(pvar single-float)** in numeric calculations executes significantly faster than code that uses other types of float pvars.

A.6 Scans and Spreads

All scan operations in Paris and in *Lisp have been enhanced to work on n-dimensional grids. In addition, two new functions have been added to *Lisp: **spread!!** and **reduce-and-spread!!**. These take advantage of new Paris instructions that spread data across the Connection Machine processors along a specified dimension.. For

definitions of **spread!!** and **reduce-and-spread!!**, see section 6.3 of the *Supplement to the *Lisp Reference Manual.*

Appendix B

Example Program 1: Text Processing

```
;;; -*- SYNTAX: COMMON-LISP; MODE: LISP; BASE: 10; PACKAGE: *LISP; -*-

(in-package '*lisp)

;;;; Author:  JP Massar.

;;;; This sample program can be found in the file
;;;;  /cm/starlisp/interpreter/f5005/text-processing-example.lisp.
;;;; Ask your systems administrator or your applications engineer
;;;; to direct you to the location of this file at your installation.

;;;; This example illustrates many features of  *Lisp Version 5.0,
;;;; including *defstruct, array pvars, 1-dimensional NEWS
;;;; communication, dynamically allocated VP sets and
;;;; communication between different VP sets.
;;;;
;;;; The object of this exercise is to read a large piece of
;;;; text into the Connection Machine system from the front end,
;;;; determine all contiguous non-blank sequences of characters ('words'),
;;;; and create a VP set which contains one word per processor.

(eval-when (compile load eval)
  (defconstant max-word-length 32)
  (defconstant max-word-length-in-bits 5)
  (deftype part-of-speech-type () `(unsigned-byte 5))
  (deftype word-length-pvar-type ()
    `(pvar (unsigned-byte max-word-length-in-bits))
    ))

;;;; Here is the definition of the parallel structure word, an instance
;;;; of which will contain our results. The part-of-speech
;;;; slot is ignored in this example.

(*defstruct word
  (characters
    (make-array max-word-length :element-type 'string-char)
    :type string
    :cm-type (vector-pvar string-char max-word-length)
```

```
    :cm-initial-value
    (make-array!! max-word-length
                  :initial-element (!! #\Space)
                  :element-type '(pvar string-char)
                  ))
  (length 0 :type (unsigned-byte 6))
  (part-of-speech 0 :type part-of-speech-type)
  )

;;;; This is the definition of WORD-VP-SET, the VP set that will contain
;;;; an instance of the parallel word structure when we are done.
;;;; We do not define how big it is at this time. That will
;;;; depend on how may words we find in the text we are
;;;; to process, so we will allocate an appropriate number
;;;; of processors during execution.

(def-vp-set words-vp-set nil
  :*defvars
  ((word-pvar (make-word!!) "" (pvar word)))
  )

;;;; Here is the main routine. It takes a piece of text
;;;; in the form of a Common Lisp string.

(defun do-text-processing (string)
  (let ((string-length (length string)))
    (*locally
      (declare (type fixnum string-length))

    ;; Allocate CHAR-VP-SET, a VP set big enough to hold all
    ;; the characters in the string, one character per processor.

      (let-vp-set
           (char-vp-set
             (create-vp-set
               (list (max *minimum-size-for-vp-set*
                          (next-power-of-two->= string-length)
                          ))))

    ;; Get into this newly-created VP set, char-vp-set,
    ;; and allocate some temporary variables we will need.

           (*with-vp-set char-vp-set
             (*let (
                    text start-word-p end-word-p space-p
                    character-position-in-word word-number
                    )
               (declare
                 (type string-char-pvar text)
                 (type boolean-pvar start-word-p end-word-p space-p)
                 (type (signed-pvar 32) character-position-in-word)
                 (type (field-pvar 32) word-number)
                 )
```

```
;; Move the string into the CM. Note that the char-vp-set
;; VP Set is 1-dimensional, but we choose to look
;; at it in terms of a 1-dimensional NEWS grid.

        (array-to-pvar-grid string text :grid-end (list string-length))
        (pppdbg text :mode :grid :end (list string-length))

;; Select those processors which contain valid characters.

        (*when (<!! (self-address-grid!! (!! 0)) (!! string-length))

;; Figure out which characters are spaces, which characters begin words,
;; and which characters end words. A word is thus defined as the
;; characters between a start bit and an end bit.

          (determine-spaces-start-and-ends
               string-length text space-p start-word-p end-word-p
               )

          (pppdbg space-p :mode :grid :end (list string-length))
          (pppdbg start-word-p :mode :grid :end (list string-length))
          (pppdbg end-word-p :mode :grid :end (list string-length))

;; Figure out the position in each word of each character,
;; and assign a unique number to each word.
;; Each character in each word will know this unique number.

          (determine-char-position-and-word-number
                space-p start-word-p
                character-position-in-word word-number
                )

          (pppdbg character-position-in-word
                 :mode :grid :end (list string-length))
          (pppdbg word-number :mode :grid :end (list string-length))
          (let ((how-many-words (*sum (if!! start-word-p (!! 1) (!! 0)))))
               (print (list 'how-many-words how-many-words))

;; Now instantiate word-vp-set, the VP set that will hold the pvar instance
;; of the parallel word structure. Give it enough processors to handle the
;; number of words that exist in the text.

            (allocate-processors-for-vp-set
              words-vp-set
              (list (max *minimum-size-for-vp-set*
                         (next-power-of-two->= how-many-words)
                         )))

;; Send the text characters from the char-vp-set to the characters array
;; in the words-vp-set, being sure each character is stored in the
;; appropiate array element, depending on its position within the word.

            (dotimes (j max-word-length)
              (*locally
                (declare (type fixnum j))
                (*when (=!! character-position-in-word (!! j))
```

```
                        (if (*or t!!)
                            (compiler-let ((*compilep* nil))
                              (*pset :no-collisions text
                                (alias!!
                                  (aref!!
                                    (alias!! (word-characters!! word-pvar))
                                  (!! j)))
                                word-number
                                :vp-set words-vp-set
                                ))))))

    ;; Now figure out how long each word is.  The array was initially filled with spaces,
    ;; so the first space we find that still exists determines how long the word is.

                        (*with-vp-set words-vp-set
                          (*setf (word-length!! word-pvar)
                                 (position!!
                                   (!! #\Space)
                                   (alias!! (word-characters!! word-pvar))))
                          (*when (minusp!! (word-length!! word-pvar))
                            (*setf (word-length!! word-pvar)
                                   (!! (the fixnum max-word-length))))
                          )

                        ;; Now print out our results in a nice format

                        (compiler-let ((*compilep* nil))
                          (dotimes (j how-many-words)
                            (let ((front-end-word (pref word-pvar j)))
                              (format t "~%Processor ~D. Length: ~D.  Word: ~A"
                                j (word-length front-end-word)
                                (word-characters front-end-word)
                                ))))

                        ))))))))

(defun determine-spaces-start-and-ends
       (string-length text space-p start-word-p end-word-p)
  (*locally
    (declare (type fixnum string-length))
    (declare (type string-char-pvar text))
    (declare (type boolean-pvar space-p start-word-p end-word-p))
    (*set space-p (char=!! text (!! #\Space)))

  ;; A character begins a word if it is the first character and it is not a space,
  ;; or it is not a space and the previous character is a space.
  ;; Since we are viewing the machine as a 1-d grid, we can use news!! to
  ;; retrieve the character value in the previous processor.
  ;; If we were viewing the machine in cube order, we would have had to
  ;; use pref!!, which is significantly slower than news!!.

    (*set start-word-p
             (and!! (not!! space-p)
                    (or!! (zerop!! (self-address-grid!! (!! 0)))
                          (char=!! (!! #\Space) (news!! text -1))
                          )))
```

```
   ;; A character ends a word if it is the last character and it is not a space,
   ;; or it is not a space and the next character is a space.

    (*set end-word-p
             (and!! (not!! space-p)
                    (or!! (=!! (self-address-grid!! (!! 0))
                               (1-!! (!! string-length)))
                          (char=!! (!! #\Space) (news!! text 1))
                          )))))

(defun determine-char-position-and-word-number
       (space-p start-word-p character-position-in-word word-number)
  (*locally
    (declare (type boolean-pvar space-p start-word-p end-word-p))
    (declare (type (signed-pvar 32) character-position-in-word))
    (declare (type (field-pvar 32) word-number))

   ;; If a character is a space, it has no position inside a word.  Use -1 to indicate this.
   ;; Otherwise, figure out the character's position from the start of the word by
   ;; using scan!! to add up 1's starting at the start bit.

    (*set character-position-in-word
          (if!! space-p
                (!! -1)
                (1-!! (scan!! (!! 1) '+!! :dimension 0 :segment-pvar start-word-p))
                ))

   ;; There are as many words as there are start bits.  Enumerate these start bits
   ;; and then tell each character in the word this unique number.

    (*when start-word-p (*set word-number (enumerate!!)))
    (*set word-number
             (scan!! word-number
                       'copy!! :dimension 0 :segment-pvar start-word-p
                       ))))
```

```
  (progn (*cold-boot) (do-text-processing "This is some text to process"))

;;;Below is the output from running the above command.

TEXT: #\T #\h #\i #\s #\Space #\i #\s #\Space #\s #\o #\m #\e
      #\Space #\t #\e #\x #\t #\Space #\t #\o
      #\Space #\p #\r #\o #\c #\e #\s #\s
SPACE-P: NIL NIL NIL NIL T NIL NIL T NIL NIL NIL NIL
         T NIL NIL NIL NIL T NIL NIL
         T NIL NIL NIL NIL NIL NIL NIL
START-WORD-P: T NIL NIL NIL NIL T NIL NIL T NIL NIL NIL
              NIL T NIL NIL NIL NIL T NIL
              NIL T NIL NIL NIL NIL NIL NIL
END-WORD-P: NIL NIL NIL T NIL NIL T NIL NIL NIL NIL T
            NIL NIL NIL NIL T NIL NIL T
            NIL NIL NIL NIL NIL NIL NIL T
```

```
CHARACTER-POSITION-IN-WORD: 0 1 2 3 -1 0 1 -1 0 1 2 3
                            -1 0 1 2 3 -1 0 1
                            -1 0 1 2 3 4 5 6
WORD-NUMBER: 0 0 0 0 0 1 1 1 2 2 2 2 2 3 3 3 3 3 4 4 4 5 5 5 5 5 5 5
(HOW-MANY-WORDS 6)
Processor 0. Length: 4.  Word: This
Processor 1. Length: 2.  Word: is
Processor 2. Length: 4.  Word: some
Processor 3. Length: 4.  Word: text
Processor 4. Length: 2.  Word: to
Processor 5. Length: 7.  Word: process
NIL

|#
```

Appendix C

Example Program 2: Determinants

```
;;; -*- SYNTAX: COMMON-LISP; MODE: LISP; BASE: 10; PACKAGE: *LISP; -*-

(in-package '*lisp)


;;;; Author:  JP Massar.

;;;; This sample code can be found in the file
;;;;  /cm/starlisp/interpreter/f5005/determinates-examples.lisp.
;;;; Ask your systems administrator or your applications engineer
;;;; to direct you to the location of this file at your installation.
;;;;
;;;; The goal of this exercise is to calculate the determinants of a
;;;; set of complex matrices. We take a three-dimensional
;;;; cubic VP set of arbitrary size and, in each processor, we
;;;; place a square complex matrix of arbitrary size. We then
;;;; calculate the determinant of each matrix in the processors
;;;; that lie along the main diagonal of the cube.  This result is copied
;;;; into all the other processors in the X-Y plane determined by the Z
;;;; coordinate of the processor that has computed the determinant.
;;;;
;;;; This code illustrates the use of complex pvars, of a mutable general
;;;; pvar that becomes an array pvar, of spread!!, of *map, and of
;;;; recursion in *Lisp.
;;;;


;;;; Define a VP set of unknown size that has a pvar, which will eventually
;;;; contain a complex matrix of unknown size in each virtual processor.

(def-vp-set complex-cube-vp-set nil
  :*defvars
  ((matrix-pvar))
  )

;;;;  This is the main routine. Here we obtain the matrix we wish to deal with,
;;;; select its main diagonal, calculate the determinate
;;;; in each of the selected processors, and spread the
;;;; result out to every processor in the same X-Y plane.
```

```
(*proclaim '(*defun determinant-of-complex-matrix!!))

(defun main (side-of-cube side-of-matrix)
  (*cold-boot)
  (get-complex-matrix side-of-cube side-of-matrix)
  (*with-vp-set complex-cube-vp-set
    (*let ((determinant (!! 0.0)))
      (declare (type single-complex-pvar determinant))
      (*when (=!! (self-address-grid!! (!! 0))
                  (self-address-grid!! (!! 1))
                  (self-address-grid!! (!! 2))
                  )
            (*set determinant (determinant-of-complex-matrix!! matrix-pvar))
            (*pset :no-collisions determinant determinant
                   (cube-from-grid-address!!
                        (!! 0) (!! 0) (self-address-grid!! (!! 2))
                        )))
      (*set determinant (spread!! determinant 0 0))
      (*set determinant (spread!! determinant 1 0))
      (ppp determinant :mode :grid :end '(4 4 2) :ordering '(2 0 1))
      )))

;;;; We first instantiate the complex-cube-vp-set to have
;;;; some defined size.  Then we create a complex matrix
;;;; of some defined square size  and make matrix-pvar contain that array.
;;;; Finally, we initialize the elements of the matrix  with random values.

(defun get-complex-matrix (side-of-cube side-of-matrix)
  (allocate-processors-for-vp-set
    complex-cube-vp-set
    (list side-of-cube side-of-cube side-of-cube)
    )
  (*with-vp-set complex-cube-vp-set
    (compiler-let ((*compilep* nil))
      (*set matrix-pvar
            (make-array!!
              (list side-of-matrix side-of-matrix)
              :element-type 'single-complex-pvar
              )))
    (let ((j 0))
      (*map
           #'(lambda (x)
               (*set (the single-complex-pvar x)
                     (complex!! (!! (the fixnum (random 5))))
                     )
               (incf j)
               )
           matrix-pvar
           ))
    ))
```

```
;;;; This proclamation tells the *Lisp compiler what type of pvar is returned
;;;; from the named function:

(*proclaim '(ftype (function (t) single-complex-pvar)
                          determinant-of-complex-matrix!!
                          recursive-determinant
                          ))

(*defun determinant-of-complex-matrix!! (complex-square-matrix)
  (declare (type (array-pvar (complex single-float) 2) complex-square-matrix))

 ;; Here, we figure out how large our matrix is on a side.

  (let ((matrix-row-size (*array-dimension complex-square-matrix 0)))

    (*let ((determinant (!! 0.0)))
      (declare (type single-complex-pvar determinant))

      (cond

    ;; A 1-element matrix is its own determinant.

          ((eql 1 matrix-row-size)
           (*set determinant (aref!! complex-square-matrix (!! 0) (!! 0)))
           )

    ;; A two-by-two matrix

    ;; A B
    ;; C D

    ;; has determinant AD - BC

          ((eql 2 matrix-row-size)
           (*set determinant
                 (-!!
                       (*!! (aref!! complex-square-matrix (!! 0) (!! 0))
                            (aref!! complex-square-matrix (!! 1) (!! 1)))
                       (*!! (aref!! complex-square-matrix (!! 1) (!! 0))
                            (aref!! complex-square-matrix (!! 0) (!! 1)))
                       )))

          (t
           (*set determinant
                 (recursive-determinant complex-square-matrix matrix-row-size)
                 ))

          )

      determinant

      )))
```

```
(defun recursive-determinant (complex-square-matrix matrix-row-size)

 ;; For more than two dimensions, we use a recursive  algorithm.
 ;; We allocate a complex matrix measuring 1 less on a side than
 ;; the original matrix.  Then, for each element of the top row
 ;;  of the original  matrix, we copy the appropriate elements into
 ;; the reduced matrix and recurse.

  (*locally
    (declare (type (array-pvar (complex single-float) 2) complex-square-matrix))

    (let ((reduced-size (1- matrix-row-size))
          (sign 1)
          )

      (*let (reduced-array determinant)
           (declare
             (type (array-pvar (complex single-float) (reduced-size reduced-size))
                   reduced-array
                   ))
           (declare (type single-complex-pvar determinant))
           (*set determinant (!! 0.0))

           (dotimes (j matrix-row-size)
             nil
             (let ((reduced-column-index 0))
               nil
               (do ((column-index 1 (1+ column-index)))
                   ((= column-index matrix-row-size))
                 nil
                 (let ((reduced-row-index 0))
                      nil
                      (dotimes (row-index matrix-row-size)
                        nil
                        (when (not (eql row-index j))
                          (*locally
                            (declare
                              (type fixnum reduced-row-index row-index))
                              (type fixnum reduced-column-index column-index))
                            (*setf (aref!! reduced-array
                                           (!! reduced-row-index)
                                           (!! reduced-column-index))
                                   (aref!! complex-square-matrix
                                           (!! row-index)
                                           (!! column-index)
                                           )))
                          (incf reduced-row-index)
                          )))
                 (incf reduced-column-index)
                 ))
```

```
            (*let ((sub-determinate
                      (determinant-of-complex-matrix!! reduced-array))
                   (multiplicand
                      (aref!! complex-square-matrix (!! (the fixnum j)) (!! 0))))
              (declare (type single-complex-pvar sub-determinate multiplicand))
              (print (list 'DETERMINANT-OF-SUBMATRIX (pref sub-determinate 0)))
              (print (list 'MUTLIPLICAND (pref multiplicand 0)))
              (print (list 'SIGN sign))
              (*set determinant
                    (+!! determinant
                         (*!! sub-determinate multiplicand (!! (the fixnum sign)))
                         )))

            (print (list 'SUBTOTAL (pref determinant 0)))
            (setq sign (* sign -1))           )

          determinant))))

  (main 8 3)
```

;;;Below is sample output from running the above command.

```
(DETERMINANT-OF-SUBMATRIX #C(8.0 0.0))
(MUTLIPLICAND #C(1.0 0.0))
(SIGN 1)
(SUBTOTAL #C(8.0 0.0))
(DETERMINANT-OF-SUBMATRIX #C(4.0 0.0))
(MUTLIPLICAND #C(4.0 0.0))
(SIGN -1)
(SUBTOTAL #C(-8.0 0.0))
(DETERMINANT-OF-SUBMATRIX #C(-2.0 0.0))
(MUTLIPLICAND #C(3.0 0.0))
(SIGN 1)
(SUBTOTAL #C(-14.0 0.0))

(2 0 1)

DIMENSION 2, COORDINATE 0

     DIMENSION 0    ----->

#C(-14.0 0.0) #C(-14.0 0.0) #C(-14.0 0.0) #C(-14.0 0.0)
#C(-14.0 0.0) #C(-14.0 0.0) #C(-14.0 0.0) #C(-14.0 0.0)
#C(-14.0 0.0) #C(-14.0 0.0) #C(-14.0 0.0) #C(-14.0 0.0)
#C(-14.0 0.0) #C(-14.0 0.0) #C(-14.0 0.0) #C(-14.0 0.0)

DIMENSION 2, COORDINATE 1

     DIMENSION 0    ----->

#C(-14.0 0.0) #C(-14.0 0.0) #C(-14.0 0.0) #C(-14.0 0.0)
#C(-14.0 0.0) #C(-14.0 0.0) #C(-14.0 0.0) #C(-14.0 0.0)
#C(-14.0 0.0) #C(-14.0 0.0) #C(-14.0 0.0) #C(-14.0 0.0)
#C(-14.0 0.0) #C(-14.0 0.0) #C(-14.0 0.0) #C(-14.0 0.0)
NIL
```

Index

Index

This index covers only the *Lisp language elements and concepts documented in the *Supplement to the *Lisp Reference Manual.* Also see the Master Index at the back of the binder. It combines this index with those for the *Lisp Reference Manual* and the *Lisp Compiler Guide.*

S

T