The
Connection Machine
System

*Lisp Dictionary

Version 6.1
October 1991

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, February 1990
Revised, October 1991

****

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

****

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142–1264
(617) 234–1000/876–1111

Contents

Part II *Lisp Dictionary

About This Manual

Objectives

The **Lisp Dictionary* is a complete reference source for the essential constructs of the *Lisp language. It is intended to provide quick access to the definitions of all *Lisp functions, macros, and global variables. It is not intended to explain the conceptual basics of programming in *Lisp, although a glossary of important and frequently used terms is included.

Note: This document reflects the *Lisp language as implemented on the Connection Machine models CM-2 and CM-200. The *Lisp Glossary, in particular, is specific to these models in its descriptions of hardware features. Connection Machine model CM-5 users should also refer to the *Porting to CM-5 *Lisp* document for differences between the two implementations.

Intended Audience

This reference dictionary is intended for readers with a working knowledge of Common Lisp, as described in *Common Lisp: The Language*, and a general understanding of the Connection Machine system. The *Getting Started In *Lisp* guide is a good source for the level of introductory information you need to use this dictionary—in particular, its appendices provide a concise overview of the CM system. The first chapter of the *CM User's Guide* is also a good source for this information, and the *Connection Machine Technical Summary* provides a more in-depth introduction to the CM, including a detailed look at how the CM operates.

Revision Information

This revised edition of the dictionary conforms with Version 6.1 of the *Lisp software, as implemented on the CM-2 and CM-200. It does *not* describe the changes implemented in Version 7.0 of the *Lisp software for the CM-5. These changes are currently documented in the manual *Porting to CM-5 *Lisp*.

Organization of This Manual

The **Lisp Dictionary* is divided into two parts.

Part I, "*Lisp Overview," provides a complete list of the functions, macros, and important global variables of *Lisp, as well as several chapters of useful overview material.

Part II, "*Lisp Dictionary," is a complete dictionary of all functions and macros in the *Lisp language.

Organization of This Manual, cont.

Part I. *Lisp Overview

Chapter 1. *Lisp Functions and Macros
A list of the names of all functions and macros in *Lisp, grouped by purpose.

Chapter 2. *Lisp Global Variables
Descriptions of the important global variables in *Lisp.

Chapter 3. *Lisp Glossary
Definitions of important terms used here and in other *Lisp manuals.

Chapter 4. *Lisp Type Declaration
A list of *Lisp data types, exact instructions for using (and *not* using) type declarations in *Lisp code, and a summary of the data type coercion rules of *Lisp.

Chapter 5. *Lisp Compiler Options
Descriptions of the effects of each of the many *Lisp compiler options.

Part II. *Lisp Dictionary
A complete dictionary of the *Lisp language, including all *Lisp functions and macros.

Related Documents

- *Getting Started In *Lisp*
  This manual provides both an overview of *Lisp and an introduction to *Lisp programming.

- *Porting to CM-5 *Lisp*
  This manual provides a summary of the changes made to the *Lisp language in Version 7.0 for the CM-5.

- *Paris Reference Manual*
  This manual describes Paris (for *parallel instruction set*), the low-level programming language of the CM-2 and CM-200. *Lisp programmers who wish to make use of Paris calls from *Lisp should refer to the Paris manual for more information.

- *CM User's Guide*
  This manual provides overview and introductory material for new users of the CM.

- *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 second edition, published in 1990, includes information about changes and extensions recommended by the ANSI technical committee X3J13 for the forthcoming ANSI standard Common Lisp.

Notation Conventions

Symbol names and code examples in running text appear in bold, as in ***cold–boot**. Code examples set off from the main text appear in a typewriter style typeface, as follows:

```
(pref a 23)
```

Names that stand for pieces of code (metavariables) appear in italics, as in *pvar-expression*. In function or macro definitions, argument names appear in italics. Keywords and argument list symbols (**&optional**, **&rest**, etc.) appear in bold:

pref *pvar-expression send-address* **&key :vp-set**

Argument names typically indicate the data type(s) accepted for that argument; for example, argument names containing the term *pvar* must be parallel variables. The name *integer-pvar* restricts an argument to a parallel variable with integer values. Functions typically signal an error when given arguments of an improper type.

The table below summarizes these notation conventions:

Convention	Meaning
boldface	Symbol names, keywords, and code examples in text.
italics	Metavariables and argument names.
`typewriter`	Code examples set off from text.
=>	Evaluates to.
==>	Expands into (macros, for example).
<=>	Are equivalent (produce the same result).

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 02142–1264
Internet Electronic Mail:	customer–support@think.com
uucp Electronic Mail:	ames!think!customer–support
Telephone:	(617) 234–4000 (617) 876–1111

For Symbolics Users Only

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

`To:` customer–support@think.com

Please supplement the automatic report with any further pertinent information.

Part I
*Lisp Overview

Chapter 1

*Lisp Functions and Macros

This chapter provides an overview of the functions and macros of *Lisp, organized in categories of functionally related operations. Only the names of functions are shown; consult the corresponding entry in the dictionary for argument lists and descriptions.

1.1 Basic Pvar Operations

*Lisp includes basic operations to allocate, access, modify, and deallocate pvars.

1.1.1 Pvar Allocation

These operations allocate/deallocate permanent pvars:

***deallocate–*defvars**	***defvar**

These operations allocate/deallocate global pvars:

allocate!!	***deallocate**

These operations allocate local pvars for the duration of a body of code:

***let**	***let***

This operation returns a temporary pvar with the same value in each processor:

!!

These operations return a temporary pvar of a specific data type:

array!! **front–end!!** **make–array!!**
typed–vector!! **vector!!**

1.1.2 Pvar Data Type Declaration and Conversion

These forms are used to declare/undeclare the data type of a pvar:

***locally** ***proclaim** **unproclaim**

These operations are used to convert pvars from one data type to another:

coerce!! **taken–as!!**

1.1.3 Pvar Referencing and Modification

This operation is used to reference the values of a pvar:

pref

These operations are used to modify the values of a pvar:

***set** ***setf**

These operations are used to define ***setf** methods for user-defined functions:

***defsetf** ***undefsetf**

This operation is used in passing aggregate pvar elements to user-defined functions, to prevent copies of those elements from being made:

alias!!

1.1.4 Pvar Information

These predicate operations test the data type of a pvar:

booleanp!!	**characterp!!**	**complexp!!**	**floatp!!**
front-end-p!!	**integerp!!**	**numberp!!**	**string-char-p!!**
structurep!!	**typep!!**		

These operations return general information about a pvar:

allocated-pvar-p	**describe-pvar**	**pvar-exponent-length**
pvar-mantissa-length	**pvar-name**	**pvarp**
pvar-plist	**pvar-type**	**pvar-vp-set**

These operations return Paris-level information about a pvar:

pvar-length	Returns Paris field length of a pvar, in bits.
pvar-location	Returns Paris field-id of a pvar.

These operations are used to print the values contained in a pvar:

ppp	**ppp!!**	**ppp-address-object**
ppp-css	**pppdbg**	**ppp-struct**
pretty-print-pvar	**pretty-print-pvar-in-currently-selected-set**	

1.2 *Lisp Function Definition

These Common Lisp operations are used to define, call, and trace *Lisp functions:

apply	**defun**	**funcall**
trace	**untrace**	

These *Lisp operations are used to define, call, and trace user-defined *Lisp functions that must reset the *Lisp stack (see the definition of ***defun** for more information):

***apply**	***defun**	***funcall**
***trace**	**un*defun**	***untrace**

1.3 Processor Selection

These forms conditionally bind the currently selected set of processors during the evaluation of their body forms or clauses:

***all**	***case**	**case!!**	***cond**
cond!!	***ecase**	**ecase!!**	***if**
if!!	***unless**	***when**	**with–css–saved**

This form iterates over the currently selected set of processors:

do–for–selected–processors

These forms return a list of the send addresses of all active processors:

list–of–active–processors **loap**

1.4 Operations on Simple Pvars

*Lisp includes specialized operations for simple (boolean, numeric, or character) pvars.

1.4.1 Boolean Logical Operators

These operations perform logical operations on boolean pvars:

and!!	**not!!**	**or!!**	**xor!!**

1.4.2 Numeric Pvar Operations

*Lisp includes operations that perform tests and math operations on numeric pvars.

Numeric Predicates

evenp!!	**minusp!!**	**null!!**
oddp!!	**plusp!!**	**zerop!!**

Relational Operators

=!!	**<!!**	**>!!**
/=!!	**<=!!**	**>=!!**
eq!!	**eql!!**	**equalp!!**

Math Operators

+!!	**–!!**	***!!**	**/!!**
1+!!	**1–!!**	**abs!!**	**ceiling!!**
compare!!	***decf**	**exp!!**	**expt!!**
floor!!	**gcd!!**	***incf**	**isqrt!!**
lcm!!	**log!!**	**max!!**	**min!!**
mod!!	**random!!**	**rem!!**	**round!!**
signum!!	**sqrt!!**	**truncate!!**	

Trigonometric Functions

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

Floating-Point Pvar Operators

fceiling!!	**ffloor!!**	**float!!**
float–sign!!	**fround!!**	**ftruncate!!**
scale–float!!		

Floating-Point Pvar Information Functions

float–epsilon!!	**least–positive–float!!**	**least–negative–float!!**
most–positive–float!!	**most–negative–float!!**	**negative–float–epsilon!!**

Complex Pvar Operators

abs!!	**cis!!**	**complex!!**
conjugate!!	**imagpart!!**	**phase!!**
realpart!!		

Bitwise Integer Operators

ash!!	**byte!!**
byte–position!!	**byte–size!!**
deposit–byte!!	**deposit–field!!**
dpb!!	**gray–code–from–integer!!**
integer–from–gray–code!!	**integer–length!!**
integer–reverse!!	**load–byte!!**
ldb!!	**ldb–test!!**
mask–field!!	**rot!!**

Bitwise Logical Operators

boole!!	**logand!!**	**logandc1!!**
logandc2!!	**lotbitp!!**	**logcount!!**
logeqv!!	**logior!!**	**lognand!!**
lognor!!	**lognot!!**	**logorc1!!**
logorc2!!	**logtest!!**	**logxor!!**

1.4.3 Character Pvar Operations

*Lisp includes operations that construct, test, and compare character pvars.

Character Pvar Operators

character!!	**char–downcase!!**	**char–flipcase!!**
char–int!!	**char–upcase!!**	**code–char!!**
digit–char!!	**int–char!!**	**make–char!!**

Character Pvar Attribute Operators

char–bit!!	**char–bits!!**	**char–code!!**
char–font!!	**initialize–character**	**set–char–bit!!**

Character Pvar Predicates

alpha–char–p!!	**alphanumericp!!**	**both–case–p!!**
characterp!!	**digit–char–p!!**	**graphic–char–p!!**
lower–case–p!!	**standard–char–p!!**	**string–char–p!!**
upper–case–p!!		

Character Pvar Comparisons

char=!!	**char<!!**	**char>!!**
char/=!!	**char<=!!**	**char>=!!**
char–equal!!	**char–greaterp!!**	**char–lessp!!**
char–not–equal!!	**char–not–greaterp!!**	**char–not–lessp!!**

1.5 Operations on Aggregate Pvars

*Lisp includes specialized operations for aggregate (array, structure, or front-end) pvars.

1.5.1 Array Pvar Operations

*Lisp includes operations to create, modify, and test multidimensional array pvars. Also included are specialized operations for one-dimensional array pvars (vectors).

Basic Array Pvar Operations

These operations return a temporary array pvar:

array!!	**make–array!!**

These operations obtain information about an array pvar:

***array–dimension**	**array–dimension!!**
***array–dimensions**	**array–dimensions!!**
***array–element–type**	**array–in–bounds–p!!**
***array–rank**	**array–rank!!**
***array–total–size**	**array–total–size!!**
array–row–major–index!!	**sideways–array–p**

These operations access elements of array pvars:

aref!!	**row–major–aref!!**
row–major–sideways–aref!!	**sideways–aref!!**

These operations map a function over a set of array pvars:

amap!!	***map**

These are specialized operations for bit-array pvars:

bit!!	**bit–and!!**	**bit–andc1!!**	**bit–andc2!!**
bit–eqv!!	**bit–ior!!**	**bit–nand!!**	**bit–nor!!**
bit–not!!	**bit–orc1!!**	**bit–orc2!!**	**bit–xor!!**
sbit!!			

These operations convert arrays to and from a sideways (slicewise) orientation:

***processorwise**	***sideways–array**	***slicewise**

Vector Pvar Operations

These operations return a temporary vector pvar:

typed–vector!!	**vector!!**

These are specialized operations for vector (one-dimensional array) pvars:

cross–product!!	**dot–product!!**	**v+!!**
v–!!	**v*!!**	**v/!!**
v+scalar!!	**v–scalar!!**	**v*scalar!!**
v/scalar!!	**vabs!!**	**vabs–squared!!**
vector–normal!!	**vscale!!**	**vscale–to–unit–vector!!**
***vset–components**		

These are serial (front-end) equivalents to the parallel vector operators:

cross–product
dot–product
v+
v–
v*
v/
v+–constant
v––constant
v*constant
v/–constant
vabs
vabs–squared
vceiling
vector–normal
vfloor
vround
vscale
vscale–to–unit–vector
vtruncate

These are specialized operations for sequence pvars:

copy–seq!!
count!!
count–if!!
count–if–not!!
every!!
***fill**
find!!
find–if!!
find–if–not!!
length!!
notany!!
notevery!!
***nreverse**
nsubstitute!!
nsubstitute–if!!
nsubstitute–if–not!!
position!!
position–if!!
position–if–not!!
reduce!!
reverse!!
some!!
subseq!!
substitute!!
substitute–if!!
substitute–if–not!!

Note that in *Lisp, sequence pvars are defined as one-dimensional array (vector) pvars.

1.5.2 Structure Pvar Operations

This operation defines a parallel structure type and defines functions that create and access instances of that parallel structure type:

***defstruct**

1.6 Processor Addressing Operations

*Lisp includes operators that provide processor addressing information.

1.6.1 Processor Enumeration, Ranking, and Sorting

This operator enumerates the currently active processors:

enumerate!!

These operators rank and sort values in the currently active processors:

rank!! **sort!!**

1.6.2 Send/NEWS Address Operators

These operators provide access to the send and grid addresses of processors:

cube–from–grid–address
cube–from–vp–grid–address
grid–from–cube–address
grid–from–vp–cube–address
self–address!!
cube–from–grid–address!!
cube–from–vp–grid–address!!
grid–from–cube–address!!
grid–from–vp–cube–address!!
self–address–grid!!

These operations are tests for off-grid processor addresses:

off–grid–border–p!!
off–grid–border–relative–p!!
off–grid–border–relative–direction–p!!
off–vp–grid–border–p!!

1.6.3 Address Object Operators

These operators create and manipulate address objects:

address–nth
address–plus–nth
address–rank
grid grid!!
self!!
address–nth!!
address–plus–nth!!
address–rank!!
grid–relative!!

1.7 Inter- and Intra-Processor Communication Operations

*Lisp provides operations that transfer values between pvars, exchange values between different processors, execute scans and reductions across processors, and perform global tests.

1.7.1 Inter-Pvar Communication Operators

These operators transfer values between pvars using global routing:

pref!! ***pset**

1.7.2 NEWS Communication Operators

These operators transfer values between pvars using NEWS communication:

***news**	**news!!**	**news–border!!**
***news–direction**	**news–direction!!**	

1.7.3 Front-End Array to Pvar Communication Operators

These operators transfer values between arrays on the front end and pvars on the Connection Machine:

array–to–pvar	**array–to–pvar–grid**
pvar–to–array	**pvar–to–array–grid**

1.7.4 Scan and Spread Operators

These operators perform scans and reductions, and spread values across processors:

reduce–and–spread!!	**scan!!**
spread!!	

1.7.5 Segment Set Scanning Operators

These operators create and manipulate segment set objects, and perform segmented scans:

create–segment–set!!
segment–set–end–bits
segment–set–end–address
segment–set–start–bits
segment–set–start–address
segment–set–processor–not–in–any–segment
segment–set–processor–not–in–any–segment!!
segment–set–scan!!
segment–set–end–bits!!
segment–set–end–address!!
segment–set–start–bits!!
segment–set–start–address!!

1.7.6 Global Communication Operators

These operators perform a global test or function, returning a single front-end value:

***and** ***integer–length** ***logand** ***logior**
***logxor** ***max** ***min** ***or**
***sum** ***xor**

1.8 VP Set Operations

These operations define, allocate, and deallocate fixed-size and flexible VP sets.

1.8.1 VP Set Definition Operators

This operation is used to define permanent VP sets, both fixed-size and flexible:

def–vp–set

These operations are used to define and allocate temporary, fixed-size VP sets:

create–vp–set **let–vp–set**

These operations are math utilities that are useful in defining the size of VP sets:

next–power–of–two–>= **power–of–two–p**

1.8.2 VP Set Geometry Functions

These operations create and deallocate the geometry objects used in defining VP sets:

create–geometry	**deallocate–geometry**

1.8.3 Flexible VP Set Allocation Operators

These operations are used to modify the geometry of a flexible VP set:

allocate–vp–set–processors	**allocate–processors–for–vp–set**
deallocate–vp–set–processors	**deallocate–processors–for–vp–set**
set–vp–set–geometry	**with–processors–allocated–for–vp–set**

1.8.4 VP Set Deallocation Operators

These operations are used to deallocate VP sets:

deallocate–def–vp–sets	**deallocate–vp–set**

1.8.5 Current VP Set Operators

These operations are used to select the current VP set and to get information about its size:

set–vp–set	***with–vp–set**
dimension–size	**dimension–address–length**

1.8.6 VP Set Operators

These operations are used to obtain information about any VP set:

describe–vp–set	**vp–set–deallocated–p**	**vp–set–dimensions**
vp–set–rank	**vp–set–total–size**	**vp–set–vp–ratio**

1.9 General Information Operations

This operator provides a limited help function for *Lisp symbols:

help

These operators trace and display the current levels of CM memory use:

***room** **trace–stack**

This macro uses the *Lisp compiler to expand a piece of *Lisp code so that you can see the resulting Lisp/Paris code:

ppme

1.10 Entertainment Operations

This operator provides access to the front-panel LED's:

***light**

1.11 Connection Machine Initialization Functions

These operators reinitialize the Connection Machine system:

***cold–boot** ***warm–boot**

These operators add and remove forms from the cold- and warm-boot initialization lists:

add–initialization **delete–initialization**

This operator toggles between the ***lisp** and **user** packages in the *Lisp interpreter and in the *Lisp simulator:

***lisp**

Chapter 2

*Lisp Global Variables

2.1 Predefined Pvars

These are permanent pvars that are predefined by *Lisp as parallel equivalents for the Common Lisp constants **t** and **nil**. It is an error to use either **t!!** or **nil!!** as the destination for ***set**, ***pset**, or any other form that modifies its argument.

This is a predefined pvar with the value **nil** in each processor:

nil!! [*Constant*]

This is a predefined pvar with the value **t** in each processor:

t!! [*Constant*]

2.2 Configuration Variables

*Lisp provides a number of configuration-dependent variables with values that are set by operators such as ***cold–boot**, **set–vp–set**, and ***with–vp–set**. A program that depends only on these configuration variables will run on a Connection Machine system in any grid configuration and at any VP ratio.

It is an error to access these variables before ***cold–boot** has been called for the first time. Also, the user must not modify the values of any of these configuration variables.

current–cm–configuration [*Variable*]

The value of this variable is a list of integers. The *n*th element of the list is the size of the *n*th dimension in the current machine configuration.

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

The value of this variable is a list of integers. The *n*th element of the list defines the number of bits necessary to hold a grid (NEWS) address coordinate for the *n*th dimension of the current VP set.

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

The value of this variable is the number of bits needed to hold the send address of a single processor in the current VP set. The variable ***log–number–of–processors–limit*** is an obsolete equivalent.

current–vp–set [*Variable*]

This variable is always bound to the current VP set. Its value changes whenever the current VP set changes. It is bound by default to the ***default–vp–set***. The operators **set–vp–set** and ***with–vp–set** can be used to change the current VP set.

default–vp–set [*Variable*]

The value of this variable is the default VP set, the VP set that is current when no other VP set is current. If no initial dimensions are specified, the first time ***cold–boot** is called, ***default–vp–set*** is bound to a two-dimensional VP set with a VP ratio of one.

log–number–of–processors–limit [*Variable*]

This obsolete variable is equivalent to the variable ***current–send–address–length***. It provides the base 2 logarithm of the number of processors attached.

minimum–size–for–vp–set [*Variable*]

The value of this variable is the minimum number of virtual processors with which a VP set may be defined. In the current implementation, this is also the number of physical processors that is currently attached. The product of the dimensions of any VP set must be greater than or equal to the value of this variable.

number–of–dimensions [*Variable*]

This variable is always bound to the number of dimensions in the current VP set. Its value changes whenever the current VP set changes.

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

This variable is always bound to the number of virtual processors in the current VP set. Its value changes whenever the current VP set changes.

2.3 Initialization List Variables

These variables each contain a set of forms that are executed automatically before and after each execution of ***cold–boot** and ***warm–boot**. The *Lisp functions **add–initialization** and **delete–initialization** are used to add and remove forms from these lists.

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

The forms in this list are executed immediately following any call to ***cold–boot**.

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

The forms in this list are executed immediately following any call to ***warm–boot**.

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

The forms in this list are executed immediately prior to any call to ***cold–boot**.

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

The forms in this list are executed immediately prior to any call to ***warm–boot**.

2.4 Configuration Limits

These constants and variables determine the size limits for specific *Lisp data types. Other than as documented here, they should not be modified in any way.

2.4.1 Array Size Limits

These constants are implementation-dependent limits on the dimension length, rank, and total size of array pvars. They should not be modified in any way.

***array–dimension–limit** *[Constant]*

This is the upper exclusive bound on the extent of a single array pvar dimension. Each dimension specified for an array pvar must be less than ***array–dimension–limit**. The value of ***array–dimension–limit** is guaranteed to be greater than or equal to 1024.

***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**. The value of ***array–rank–limit** is guaranteed to be greater than or equal to 8.

***array–total–size–limit** *[Constant]*

This is the upper exclusive bound on the product of all the dimensions specified for an array pvar. The total number of elements a parallel array can have must be less than ***array–total–size–limit**. The value of ***array–total–size–limit** is guaranteed to be greater than or equal to 1024.

2.4.2 Character Attribute Size Limits

These variables represent user-specified limits on the length and value of the code, bits, and font attributes of character pvars. These variables may be set to values other than the defaults by calling the *Lisp function **initialize–character**. The value of these variables should not be modified by the user in any other way.

Note that if the **initialize–character** function is used, it must be called immediately prior to calling ***cold–boot**, because the values of the attribute variables below are used in initializing *Lisp and the Connection Machine system.

***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–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–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.

***character-limit** [*Variable*]

This is the upper exclusive bound restricting the integer value contained by a pvar of type character.

2.5 Error Checking

These variables control the error-checking measures taken by the *Lisp interpreter and compiler in evaluating and compiling code. These variables may be freely modified by the user to contain any of the specified legal values.

interpreter-safety [*Variable*]

This variable determines the amount of run-time error checking performed by the *Lisp interpreter. 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	Reserved for future expansion, do not use.
3	maximum run-time error checking; error messages emitted immediately.

safety [*Variable*]

This variable determines the amount of error-checking code generated by the *Lisp compiler. The value of ***safety*** must be an integer between 0 and 3, inclusive. The effect of each setting is given below.

0	Low safety. Error conditions are prevented from being signalled.
1	Error conditions are signalled, but notification of an error does not occur at the time the error takes place.
2	Identical to a ***safety*** level of 3 or 1, depending on the value (**t** or **nil**) of the variable ***immediate-error-if-location***, modifiable at run time.
3	High safety. Errors signalled immediately, with detailed error messages.

immediate–error–if–location *[Variable]*

Determines the action taken at run-time by code compiled with a ***safety*** value of 2. If the value of this variable is **t**, such code behaves as if compiled with a ***safety*** value of 3. If the value of this variable is **nil**, such code behaves as if compiled with a ***safety*** value of 1.

warning–level *[Variable]*

This variable controls the type of warnings generated by the *Lisp compiler. The value of ***warning–level*** must be one of the symbols **:high**, **:normal**, or **:none**. The effect of each setting is given below.

:high	Detailed warnings emitted whenever a section of code is not compiled.
:normal	Warnings generated only for invalid arguments and type mismatches.
:none	Prevents generation of any warnings.

2.6 *Lisp Compiler Code-Walker

slc:*use–code–walker* *[Variable]*

This boolean variable controls whether the code-walker portion of the *Lisp compiler is active. For more information about the code-walker, see the **Lisp Release Notes* Version 5.2. For more information about compiling *Lisp code, see the **Lisp Compiler Guide* Version 5.2.

2.7 Pretty-Printing Defaults

These variables provide global defaults for the keyword arguments of all of the pvar pretty printing operations. Some functions do not include keywords that correspond to all these global variables; consult the dictionary definition of each printing function for a list of the keyword defaults used.

ppp-default-mode *[Variable]*

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

ppp-default-format *[Variable]*

This variable provides the default value for the **:format** keyword argument. Its initial value is the string "**~s** ".

ppp-default-per-line *[Variable]*

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

ppp-default-start *[Variable]*

This variable provides the default value for the **:start** keyword argument. Its initial value is zero.

ppp-default-end *[Variable]*

This variable provides the default value for the **:end** keyword argument. Whenever the current VP set changes and whenever ***cold-boot** is called, ***ppp-default-end*** is reset to the current value of ***number-of-processors-limit***.

ppp-default-title *[Variable]*

This variable provides the default value for the **:title** keyword argument. Its initial value is **nil**, indicating that no title should be printed.

ppp-default-ordering *[Variable]*

This variable provides the default value for the **:ordering** keyword argument. Its initial value is **nil**, indicating that no special grid dimension ordering is required.

ppp-default-processor-list *[Variable]*

This variable provides the default value for the **:processor-list** keyword argument. Its initial value is **nil**, indicating that all processors between **:start** and **:end** should be displayed.

Chapter 3

*Lisp Glossary

This chapter contains a glossary of special terms and concepts used in descriptions of the *Lisp language.

3.1 Connection Machine Terminology

These are terms directly relating to the Connection Machine and its relationship to the *Lisp language.

3.1.1 Machines

Connection Machine The Connection Machine (CM) consists of a large number of *processors* that operate on data in parallel, linked together by an internal communications network and controlled by an external *front-end* computer.

front end The external computer system that transmits instructions and data to the processors of the CM and receives data returned by the processors as a result of their operations is called the *front end.*

3.1.2 Processors

processors
The conceptual entities that operate on data in parallel within the CM are called *processors*. Each processor has an associated local memory, within which data is stored and manipulated. Each processor is also connected to all other processors by an internal communications network. The term "processors" can be used to refer to the *physical processors* of the CM, but it is most commonly used to refer to the *virtual processors* simulated by the machine. This is the convention observed in this document.

physical processors
The single-bit processing units within the CM that operate on data in parallel are called the *physical processors* of the machine. Each physical processor simulates the actions of one or more *virtual processors*.

virtual processors
The conceptual processing entities simulated by the physical processors of the CM are called *virtual processors*. This simulation is transparent to the user. No matter how many virtual processors are simulated, each has its own associated memory and operates independently of the others.

active processors
Each processor maintains an internal flag that determines whether it is *active*, that is, whether or not it executes the instructions it receives. Only the active processors of the CM execute any given operation.

currently selected set
The set of all currently active CM processors is called the *currently selected set*. The currently selected set is changed by using *Lisp special forms such as ***all**, ***when**, ***if**, ***cond**, and ***case**.

3.1.3 Fields

field
Data is stored on the CM in *fields*. A field consists of a contiguous set of bits at the same location in the memory of each processor.

allocation/ deallocation
A field is created by *allocating*, or reserving, the same number of bits in the memory of each processor. When a field is no longer needed, it can be *deallocated*, freeing the memory for use in other fields.

value of a field
The value of a field in any given processor is simply the value contained in the set of bits allocated for the field in that processor's memory.

3.1.4 Connection Machine Memory

heap/stack Fields are allocated in two areas of memory on the CM known as the *heap* and the *stack*. Fields allocated on the heap are permanent, and persist until the user explicitly deallocates them. Fields allocated on the stack are temporary, and are automatically deallocated whenever the stack is cleared.

cold boot The Connection Machine operation that resets the internal state of the machine and clears its memory is called a *cold boot*. All Connection Machine fields are deallocated during a cold boot.

warm boot The Connection Machine operation that resets the internal state of the machine and clears the stack, but does not clear the heap, is called a *warm boot*. Fields allocated on the stack are deallocated during a warm boot.

3.2 *Lisp Terminology

These are terms relating to the data structures and operations of the *Lisp language.

3.2.1 Parallel Variables (Pvars)

parallel variable The *Lisp data structure that represents a collection of values stored one-per-processor on the CM is called a *parallel variable*, or *pvar*. A pvar consists of a field allocated on the CM and a front-end data structure that contains the location, length in bits, and data type of that field.

value of a pvar In any given processor, the *value* of a pvar is simply the value of its associated field in that processor.

corresponding value Given two pvars, *A* and *B*, for the value of *A* in any processor there is a *corresponding value* of *B* located in the memory of the same processor. Operations on pvars typically act by combining the corresponding values of two or more pvars.

scalar value A front-end data type, such as an integer, a character, or a structure object, is called a *scalar value*.

pvar contents The *contents* of a pvar is the entire set of scalar values stored in the field of that pvar.

Pvar Classes

There are two main classes of pvars, *heap pvars* and *stack pvars*, corresponding to the two types of Connection Machine memory.

heap pvars Heap pvars are relatively permanent, long-term storage locations for data, with global scope and dynamic extent. Heap pvars are divided into *permanent pvars* and *global pvars*.

stack pvars Stack pvars are temporary storage locations for data, with lexical scope and dynamic extent. They are automatically deallocated whenever the stack is cleared. Stack pvars are divided into *local pvars* and *temporary pvars*.

permanent pvars Permanent pvars are created by the ***defvar** macro. They are named global pvars and are automatically reallocated whenever the CM is cold-booted, unless explicitly deallocated by the user.

global pvars Global pvars are created by the **allocate!!** function. They are identical to permanent pvars, with the exception that global pvars are not reallocated when the CM is cold booted.

local pvars Local pvars are created by the ***let** and ***let*** macros. They are allocated on the stack as local variables for the duration of a body of code.

temporary pvars Temporary pvars are returned by most functions and macros in *Lisp. They are temporary storage locations intended to contain values only until those values are copied to pvars of one of the above classes. It is an error to attempt to modify any temporary pvar value.

Pvar Types

Heap and stack pvars are divided into three groups based on the data types of their values: *simple pvars*, *aggregate pvars*, and *general pvars*. Simple and general pvars may also be declared as *mutable pvars*.

simple pvars Simple pvars contain either boolean, numeric, or character values.

aggregate pvars Aggregate pvars contain either arrays, structure objects, or pointers to front-end data structures.

general pvars General pvars can contain values of differing data types, with the exception that general pvars may not contain aggregate data objects such as arrays or structures. General pvars are not as efficient as simple or aggregate pvars, because type-checking overhead is required by their use and because code containing general pvars cannot be compiled.

mutable pvars Mutable pvars are simple or general pvars that have been declared to contain values of unspecified bit sizes. *Lisp code containing simple mutable pvars cannot be compiled as efficiently as code containing simple pvars of fixed size.

3.2.2 Processor Addressing

The value of a pvar in any processor may be accessed and modified. To do this, it is necessary to specify a processor's address within the CM. There are two basic schemes in *Lisp for assigning addresses to processors: *send addressing* and *grid addressing*.

configuration An abstract arrangement of processors that groups them in an *n*-dimensional array, such as a line, a plane, or a cube, is called a *configuration*. The number of dimensions in a configuration is the *rank* of that configuration. The geometry of the current VP set determines the current configuration. Note: the terms *grid*, *machine configuration*, and *NEWS grid* are sometimes used synonymously with *configuration*.

send address Each processor has a unique *send address*, roughly corresponding to the location of the processor within the hardware. Send addresses range between zero and one less than the total number of processors. (In previous versions of *Lisp, this was referred to as the *cube address* of the processor.)

grid address A list of coordinate integers that specify a processor's position in a given configuration is called that processor's *grid address*. The number of coordinates in a grid address must be equal to the rank of the configuration. For example, the grid address of a processor in a two-dimensional configuration is a list of two integers.

address object An *address object* is a data structure that can be used as a send address but that specifies a given processor's grid address. Address objects are more flexible than grid addresses because they automatically translate grid addresses between different processor configurations. This flexibility is obtained at the cost of efficiency, however; address objects are less efficient than other forms of processor addressing.

3.2.3 Virtual Processor Sets

geometry A *geometry* is a description of the size and shape of a particular configuration of virtual processors. It can be either a list of integers or a *geometry object*.

geometry object A *geometry object* is a front-end data structure that contains a specified geometry. It is used to define the size and shape of *virtual processor sets*.

virtual processor set A *virtual processor set*, or *VP set*, is an arrangement of virtual processors in a specified *n*-dimensional geometry. A VP set can have pvars associated with it, and values may be transferred between pvars associated with different VP sets. Only one VP set, known as the *current VP set*, may be active at any given time.

VP set object A front-end data structure defining the geometry and associated pvars of a virtual processor set is called a *VP set object*.

VP ratio The number of virtual processors simulated by each physical processor on the CM for a given VP set is referred to as the *virtual processor ratio*, or *VP ratio*, of the VP set.

Classes of VP Sets

There are two main classes of VP sets, *permanent* and *temporary*. Permanent VP sets are further divided into *fixed-size* and *flexible* VP sets.

permanent VP set A *permanent VP set* is defined using the **def–vp–set** operator. Permanent VP sets are automatically reallocated when the CM is cold booted until the user explicitly deallocates them. Permanent VP sets can be either *fixed-size* or *flexible.*

temporary VP sets A temporary VP set is defined using either the **create–vp–set** or the **let–vp–set** operator. They are deallocated during a cold boot, as are their associated pvars. Temporary VP sets are always *fixed-size.*

fixed-size VP set A *fixed-size VP set* has a specific geometry that does not change. Fixed-size VP sets are defined by calling **def–vp–set** with specific geometry information.

flexible VP set A *flexible VP set* has no geometry initially—its shape and size is determined by the user at run-time. Flexible VP sets are defined by calling **def–vp–set** without providing specific geometry information. Flexible VP sets must be *instantiated* before they can be used (see below).

defined A permanent VP set (fixed or flexible) is *defined* by the **def–vp–set** operator. A temporary VP set is defined either by the **create–vp–set** or the **let–vp–set** operator.

instantiated Fixed-size VP sets can be used immediately. Flexible VP sets must be *instantiated* (assigned a temporary geometry) by an operator such as **allocate–processors–for–vp–set** before they can be used.

3.2.4 Important VP Sets

current VP set At any one time, there is one active VP set: the *current VP set.* Only pvars associated with this VP set are directly accessible, and unless otherwise specified, newly declared pvars are associated with the current VP set. The variable ***current–vp–set*** is always bound to the current VP set.

current configuration The rank and size of the current VP set, i.e., the size and shape of the set of processors currently in use, is often referred to as the *current configuration* of the machine.

default VP set When the CM is cold booted for the first time, a *default VP set* is created. Until some other VP set is created and selected, the default VP set remains current and determines the configuration of the CM. The variable ***default–vp–set*** is always bound to the default VP set.

3.3 Background Terminology

The naming convention for *Lisp operators, along with other useful background information, is described here.

!! Functions that have names ending in **!!** (pronounced *bang-bang*) return a pvar result. The **!!** is intended to resemble "||", the mathematical symbol for parallelism. **Note:** These functions return temporary pvars that may be reclaimed whenever the *Lisp stack is cleared; these temporary pvars must be copied into a more permanent class of pvar (by ***set**, for example) if you want to keep them.

***** Functions and macros with names ending in * (pronounced *star*), perform parallel operations but do not always return a pvar. The name of the language itself, "*Lisp" (*star-Lisp*), comes from this convention.

parallel equivalent of This phrase is used to describe the correspondence between a Common Lisp function and a *Lisp function function that perform similar operations. For example, **mod!!** is the parallel equivalent of Common Lisp's **mod**. This means that **mod!!** performs the same calculation as **mod**, but that **mod!!** takes parallel variables as arguments and performs the **mod** operation for each active processor within the CM.

Chapter 4

*Lisp Type Declaration

This chapter describes the different types of parallel variables, or *pvars*, available in *Lisp, discusses type declaration and the rules of type coercion, and explains how to use type declarations in *Lisp.

4.1 Pvar Types

A pvar is defined by the kind of values that can be stored in it. The following pvar types are supported in *Lisp:

general	**front–end**	**boolean**	**signed–byte**
unsigned–byte	**defined–float**	**complex**	**character**
string–char	**array**	**structure**	

For most pvar types, *Lisp provides several equivalent forms that may be used in declarations. For instance, for almost 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 <=> is equivalent and may be used interchangeably within ***proclaim**, **declare**, and **the** forms, as well as with the operators **coerce!!** and **taken–as!!**.

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

```
(pvar t)    <=>   general-pvar
```

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

```
(pvar front-end)   <=>   front-end-pvar
```

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

```
(pvar boolean)  <=>  boolean-pvar
```

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

```
(pvar (unsigned-byte width)) <=> (unsigned-pvar width)
      <=>   (unsigned-byte-pvar width)
      <=>   (field-pvar width)

(pvar bit)      <=>  (pvar (unsigned-byte 1))
```

signed-byte — A signed integer for each processor.

```
(pvar (signed-byte width)) <=> (signed-pvar width)
      <=> (signed-byte-pvar width)

(pvar fixnum)   <=>  (pvar (signed-byte fixnum-length))
                <=>  fixnum-pvar

(pvar integer)  <=>  (pvar (signed-byte *))
```

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

```
(pvar (defined-float significand-length exponent-length))
      <=> (float-pvar significand-length exponent-length)

(pvar short-float)  <=>  (pvar (defined-float 15 8))
                    <=>  short-float-pvar

(pvar single-float) <=>  (pvar (defined-float 23 8))
                    <=>  single-float-pvar

(pvar double-float) <=>  (pvar (defined-float 52 11))
                    <=>  double-float-pvar

(pvar long-float)   <=>  (pvar (defined-float 74 21))
                    <=>  long-float-pvar

(pvar float)        <=>  (pvar (defined-float * *))
                    <=>  float-pvar
```

character — A Common Lisp character for each processor.

```
(pvar character)    <=>  character-pvar
(pvar string-char)  <=>  string-char-pvar
```

complex — A complex number for each processor.

```
(pvar (complex (defined-float significand exponent)))
      <=>    (complex-pvar significand exponent)

(pvar (complex short-float))
      <=>    (pvar (complex (defined-float 15 8)))
      <=>    short-complex-pvar

(pvar (complex single-float))
      <=>    (pvar (complex (defined-float 23 8)))
      <=>    single-complex-pvar

(pvar (complex double-float))
      <=>    (pvar (complex (defined-float 52 11)))
      <=>    double-complex-pvar

(pvar (complex long-float))
      <=>    (pvar (complex (defined-float 74 21)))
      <=>    long-complex-pvar

(pvar complex)
      <=>    (pvar (complex (defined-float * *)))
      <=>    complex-pvar
```

array — A Common Lisp array for each processor.

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

(pvar (string length))   <=>    (string-pvar length)
      <=>    (pvar (vector string-char length))

(pvar (bit-vector length)) <=> (bit-vector-pvar length)
      <=>    (pvar (vector (unsigned-byte 1) length))
```

structure — A Common Lisp structure for each processor.

```
(pvar structure-name)  <=> structure-name-pvar
```

Note: *structure–name* must be a parallel structure type defined by ***defstruct**.

*Lisp allows *mutable* pvar types (pvars of varying bit–length). The most flexible type of pvar in *Lisp is the *general mutable* pvar. Mutable pvars and the general mutable pvar type are described in separate sections later in this chapter.

4.2 Using Type Declarations

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 only compile *Lisp code that references pvars that are declared to be of a definite type. (For this reason, code that uses general or mutable pvars generally will not compile.)

This section provides a basic guide to the methods and use of type declaration in *Lisp. It includes a description of the operators used for type declaration, along with a set of guidelines for the use of type declarations in user code.

Type declarations represent promises made by you to the compiler that only values of the declared type will be assigned to a variable or returned by a declared form. Type declarations do *not* cause type coercion. It is an error for a program to violate a type declaration, and the results of an incorrectly declared expression are not defined. Also, if a type declaration is changed, all compiled code that depends on that declaration must be recompiled.

4.2.1 *Lisp Declaration Operators

Three operators are used for type declaration in *Lisp: the Common Lisp declaration operators **declare** and **the**, and the *Lisp declaration operator ***proclaim**. A general description of the use of each of these operators appears below.

The ***proclaim** operator is used in the following ways:

- To declare the data type of a permanent pvar defined by ***defvar**, as in

  ```
  (*proclaim '(type (pvar single-float) my-pvar))
  (*defvar my-pvar (random!! 1.0))
  ```

  which declares the permanent pvar **my-pvar** to be of type **(pvar single-float)**.

- To declare the pvar data type returned by a user-defined *Lisp function, as in

  ```
  (*proclaim
      '(ftype (function (pvar pvar) (field-pvar 16))
           my-pvar-function))
  ```

  which declares that the pvar returned by the function **my-pvar-function** is of type **(field-pvar 16)**.

- To declare the data type of scalar variables and user-defined functions that are used in a pvar expression (any expression that returns a pvar as its value), as in the following examples:

  ```
  (*proclaim '(type (unsigned-byte 8) *my-limit*))
  (defvar *my-limit* 20)
  (*set data-pvar
      (+!! (random!! *my-limit*) (random!! *my-limit*)))
  ```

  the global variable ***my-limit*** used in the two calls to **!!** is declared to be of type **(unsigned-byte 8)**.

  An example of a function declaration is given by the expressions

  ```
  (*proclaim '(ftype (function () fixnum) die-roll))
  (defun die-roll () (+ (random 6) (random 6) 2))
  (*set dice-pvar (die-roll))
  ```

  in which the user-defined function **die-roll** is declared to return a **fixnum** result.

  Important: Do not use ***proclaim** to declare the returned values of Common Lisp functions. Instead, use the Common Lisp **the** operator as shown in the section on **the** below.

- To declare that a user-defined *Lisp function will be defined with ***defun**:

  ```
  (*proclaim '(*defun fn))
  (*proclaim '(ftype (function (t t) single-float-pvar fn))
  (*proclaim '(type single-float-pvar z)) (*defvar z)
  (defun bar () (*set z   (fn 3.0 4.0)))
  (*defun fn (a b)
      (declare (type single-float-pvar a b))
      (+!! a b))
  ```

  This is important because ***defun** operators are really macros, not functions, so if a ***defun** operation is referenced before it is defined (as in a file of *Lisp code), the "forward references" to the operator will be compiled incorrectly.

The Common Lisp **declare** operator is used in the following ways:

- To declare the pvar data type of local pvars created by ***let** or ***let***, as in

  ```
  (*let ((pvar-1 (random!! 1.0)) (pvar-2 (random!! 10)))
      (declare (type single-float-pvar pvar-1))
      (declare (type (field-pvar 8) pvar-1))
      (pvar-computation pvar-1 pvar-2))
  ```

- To declare the data types of arguments to functions defined by **defun** or ***defun**. For example,

```
(*defun pvar-computation (pvar-1 pvar-2)
    (declare (type single-float-pvar pvar-1))
    (declare (type (field-pvar 8) pvar-2))
    (combine-pvars pvar-1 pvar-2))
```

- To declare the data types of scalar local and looping variables, as in

```
(let ((limit (+ 2 (random 8))))
    (declare (type fixnum limit))
    (*let ((sum-pvar 0))
      (do((i 0 (+ i 2)))
              ((>= i limit) sum-pvar)
              (declare (type fixnum i))
              (*set sum-pvar
                (+!! sum-pvar (random!! i)
                              (random!! limit))))))
```

The Common Lisp **the** operator is used to declare the data type of an expression in situations not covered by either of the above two operators.

- To declare the data type returned by a Common Lisp expression, as in

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

- To make "on the spot" declarations where a single inline declaration is preferable to a more global, widespread declaration. For example,

```
(*set data-pvar
         (log!! (the double-float-pvar figures-pvar)))
(*set (the (pvar unsigned-byte 16) data-pvar)
      (the (pvar (unsigned-byte *))
           (if store-three-pvar-p 3 0)))
```

Note that it is no less efficient to use ***proclaim** or **declare** in place of **the** wherever this is possible, i.e., in declaring the data types of pvars and the data types returned by user-defined *Lisp functions. Readability and maintainability of code can often be improved by doing so.

4.2.2 Basic Rules of Type Declaration

The following is a set of basic guidelines for the declaration of *Lisp data objects. These rules describe the data objects that must be declared in order to permit code to compile, and describe how these objects should be declared. These rules also describe which data objects should *not* be declared.

Declaring Pvars

- Declare with ***proclaim** the data type of permanent pvars defined by ***defvar**.
- Declare with **declare** or **the** the data type of global pvars created by **allocate!!** wherever these pvars are used.
- Declare with **declare** the data type of local pvars defined by ***let** and ***let***.
- **Don't declare** the pvar data type of temporary pvars returned by **!!**.

Declaring Pvar Functions

- Declare with **declare** the arguments of a user-defined *Lisp function (i.e., a function defined by either **defun** or ***defun**).
- Declare with ***proclaim** the returned value of a user-defined *Lisp function.
- Declare with ***proclaim** all ***defun** definitions prior to all type declarations for and calls to these definitions.
- **Don't declare** the pvar data type returned by any predefined *Lisp operator.

Declaring Scalar Expressions

- Declare with ***proclaim** the data type of any scalar global variable that is used in a pvar expression.
- Declare with **declare** the data type of any scalar local variable that is used in a pvar expression (i.e., a variable defined by **let**, **let***, or the **do** family of looping operators).
- Declare with **the** the data type of any scalar expression other than a variable (i.e., a call to a Common Lisp function) that is used in a pvar expression.
- **Don't declare** the data type of scalar constants used in pvar expressions.

The next three sections provide examples for each of these rules.

Declaring Pvars

- Declare with ***proclaim** the data type of permanent pvars defined by ***defvar**. For example, the *Lisp forms

```
(*proclaim '(type (pvar (unsigned-byte 8)) perm-pvar))
(*defvar perm-pvar (random!! 255))
(*proclaim '(type boolean-pvar y-or-n-p-pvar))
(*defvar y-or-n-p-pvar (zerop!! (random!! 2)))
```

  declare **perm-pvar** to be of type **(pvar (unsigned-byte 8))**, and **y-or-n-p-pvar** to be of type **boolean-pvar**.

- Declare with **declare** or **the** the data type of global pvars created by **allocate!!** wherever these pvars are used. For example, in

```
(setq a-pvar (allocate!! 0.0 nil 'single-float-pvar))
(*set (the single-float-pvar a-pvar) (random!! 10.0))
(dotimes (i 3)
  (*incf data-pvar (the single-float-pvar a-pvar))))
```

  the **the** operator is used to declare **a-pvar** to be of type **single-float-pvar**.

  Another example is

```
(defvar pvars nil)
(dotimes (i 10)
    (push (allocate!! 0.0 nil 'single-float-pvar) pvars))
(defun randomize-nth-pvar (n)
    (*set (the single-float-pvar (nth n pvars))
         (random!! 1.0)))
```

  in which **the** is used to declare whichever allocated pvar is selected from the **float-pvars** list to be of type **single-float-pvar**.

- Declare with **declare** the data type of local pvars defined by ***let** and ***let***.

  For example,

```
(*let ((local-pvar (random!! 32)))
    (declare (type (unsigned-byte-pvar 8) local-pvar))
    (*!! (+!! local-pvar local-pvar) 2))
(*let* ((float-pvar (random!! 5.0))
        (integer-pvar (floor!! float-pvar)))
    (declare (type short-float-pvar float-pvar))
    (declare (type (field-pvar 6) integer-pvar))
    (abs!! (-!! float-pvar integer-pvar)))
```

- **Don't declare** the pvar data type of temporary pvars returned by **!!**.

  For example, the following declarations are unnecessary:

```
;;; These declarations are unnecessary.
(the (unsigned-byte-pvar 5) (!! 3))
(the character-pvar (!! #\C))
(the (array-pvar single-float (3)) (!! #(1.0 2.0 3.0)))
```

Declaring Pvar Functions

- Declare with **declare** the arguments of a user-defined *Lisp function (i.e., a function defined by either **defun** or ***defun**).

  For example, in

```
(*defun global-range (argument-pvar)
    (declare (type (field-pvar 256) argument-pvar))
    (- (*max argument-pvar) (*min argument-pvar)))
```

the **argument-pvar** to **global-range** is declared to be of type **(field-pvar 256)**, and in

```
(defun zero-pvar-when (test-pvar float-pvar)
    (declare (type boolean-pvar test-pvar))
    (declare (type double-float-pvar float-pvar))
    (if!! test-pvar float-pvar (!! 0.0)))
```

the **test-pvar** argument is declared to be of type **boolean-pvar**, and the **float-pvar** argument of type **double-float-pvar**.

- Declare with ***proclaim** the returned value of a user-defined *Lisp function.

  For example, in

```
(*proclaim
    '(ftype (function (pvar pvar) (pvar single-float))
        surface-area!!))
```

the function **surface-area!!** is declared to return a **(pvar single-float)** value.

- Declare with ***proclaim** all ***defun** definitions prior to all type declarations and calls to these operations. This is important because ***defun** operators are really macros, not functions, so if a ***defun** operation is referenced before it is defined (as in a file of *Lisp code), the "forward references" to the operator will be compiled incorrectly.

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

(*proclaim
'(ftype (function (t t) (pvar single-float)) xyzzy-foo))

(*defun xyzzy-foo (a b)
    (declare (type single-float-pvar a b))
    (+!! a b))
```

- **Don't declare** the pvar data type returned by any predefined *Lisp operator.

  For example, the following declarations are unnecessary:

```
;;; These declarations are unnecessary.
(*proclaim '(function evenp!! (t t) (pvar boolean)))
(*proclaim '(ftype (function (t) boolean-pvar) evenp!!))
(*set data-pvar (the single-float-pvar (log!! (!! 3))))
```

Declaring Scalar Expressions

- Declare with ***proclaim** the data type of any scalar global variable that is used in a pvar expression. For example, in

```
(*proclaim '(type single-float global-variable))
(defvar global-variable 50)
(*set data-pvar (log!! (!! global-variable)))
```

  the **global-variable** used to initialize **data-pvar** is declared to be a **single-float**.

  In the expression

```
(*proclaim '(type character special-char))
(defvar special-char #\Return)
(*if (char=!! char-pvar (!! special-char))
     (handle-special-char char-pvar)
     (handle-normal-char char-pvar))
```

  the variable **special-char** is declared to be of type **character**. Note that the ***proclaim** operator must be used instead of Common Lisp's **proclaim**. Otherwise, the *Lisp compiler will not have access to these declarations.

- Declare with **declare** the data type of any scalar local variable that is used in a pvar expression (i.e., a variable defined by **let**, **let***, or the **do** family of looping operators). For example, in

  ```
  (do ((i 1 (* i 2)))
      ((> i 256) data-pvar)
    (declare (type fixnum i))
    (*incf (data-pvar (!! i))))
  ```

  the iteration variable **i** is declared to be of type **fixnum**.

  Another example is the expression

  ```
  (let ((maximum-limit 10)
        (minimum-limit 2.5))
      (declare (type fixnum maximum-limit))
      (declare (type single-float minimum-limit))
      (*set condition-pvar
      (cond!! ((>!! highest-reading-pvar (!! maximum-li-
  mit))
               (front-end-pvar!! 'TOO-HIGH))
              ((<!! lowest-reading-pvar (!! minimum-limit))
               (front-end-pvar!! 'TOO-LOW))
              (t!! (front-end-pvar 'WITHIN-LIMITS)))))
  ```

  in which the local variables **maximum–limit** and **minimum–limit** are declared to be of type **fixnum** and type **single–float**, respectively.

  Important: Because the iteration variable of **dotimes** is always of type **fixnum**, it is unnecessary to use **declare** to declare the type of this variable. For example,

  ```
  ;;; The declaration in this dotimes call is unnecessary.
  (dotimes (i 50) (*incf data-pvar (!! (the fixnum i))))
  ```

- Declare with **the** the data type of any scalar expression other than a variable (i.e., a call to a Common Lisp function) that is used in a pvar expression.

  For example, in

  ```
  (*proclaim '(type fixnum sum elements))
  (*set data-pvar (the short-float (/ sum elements)))
  ```

  the expression **(/ sum elements)** is declared to be of type **short–float**.

In the expression

```
(*proclaim '(type fixnum total))
(*set data-pvar (+!! (the fixnum (+ total 4))
                     (the fixnum (- total 4))))
```

the expressions **(+ total 4)** and **(– total 4)** are declared to be of type **fixnum**.

Note that all variables used in these scalar expressions must also be declared, as shown in this example.

- Don't declare the data type of scalar constants used in pvar expressions.

  For example, the following declarations are unnecessary.

```
;;; The declarations in these forms are unnecessary.
(*set pi-pvar (!! (the short-float 3.14159)))
(*set space-char-pvar (!! (the character #\Space)))
(*set array-pvar (!! (the (array fixnum (5))
                          #(1 2 3 4 5))))
```

4.3 General Pvars

This section describes the **general** pvar data type in more detail.

(pvar t)

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 any pvar type except the **array** or **structure** types.

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.

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 general pvar source 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 a ***set** with a general pvar destination does *not* have a general pvar source, the ***set** operation depends on the type of the source pvar, as described under each pvar type in Section 4.6, "Rules of *Lisp Type Declaration and Coercion," below.

4.4 Mutable Pvars

Pvars may be declared to be *mutable*, which allows them to contain data of varying size and type. To declare a pvar as mutable, specify the symbol * in place of one or more parameters in the type specification of the pvar. For example,

```
(*let (mutable-signed-pvar)
  (declare (type (signed-pvar *) mutable-signed-pvar))
   ...)

(*proclaim '(type (pvar (defined-float * *))
                  mutable-float-pvar))
(*defvar mutable-float-pvar)
```

4.5 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.*

This is the form used within declarations to explicitly declare a mutable general pvar:

(pvar *)

For example, the following forms proclaim **random-mutable-pvar** to be a mutable general pvar and then allocate the pvar **random-mutable-pvar.**

```
(*proclaim '(type (pvar *) random-mutable-pvar))
(*defvar random-mutable-pvar)
```

If a mutable general pvar is void and a pvar of any specific data type is ***set** into it, then the mutable general pvar will assume the characteristics of that type, but will retain its status as a mutable general pvar. Once a mutable general pvar has contained data of two or more distinct types, however, it loses its mutable quality and becomes an ordinary 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).**

For the purpose of this definition, the following groups of pvar types are considered as distinct with respect to their effect on a mutable general pvar:

boolean
signed-byte and **unsigned-byte**
character and **string-char**
defined-float
complex

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-byte** data stored in it, it changes into a mutable general pvar of type **signed-byte.**

This is significant because if a mutable general pvar has held only one distinct 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 have held only one distinct type of data is much faster than the execution speed of the same code using general pvars.

Given these distinctions in type membership, so long as no data of a different type is ***set** into a mutable general pvar, the mutable general pvar will behave exactly as though it was a mutable pvar of the same type as the data last stored it.

Aggregate (array and structure) pvars are a special case. Aggregate pvars may only be ***set** into a mutable general pvar if the mutable general pvar is void. In this case, the mutable general pvar ceases to be a mutable general pvar and becomes an aggregate pvar of the same type and size as the source pvar.

4.6 Rules of *Lisp Type Declaration and Coercion

This section defines the *Lisp rules of type declaration and coercion. For each *Lisp pvar type listed below, the following questions are answered:

- Can pvars of this type be declared mutable?
- What types of data can be stored into a pvar of this type?
- What type coercions take place if the data is not of the same type as the pvar?
- What happens when data of this type is stored in a general pvar?

In each case, the latter two questions are answered by explaining the type coercions that occur when ***set** is used to copy a pvar of one type into a pvar of another type. Coercions performed by other *Lisp operators (such as **coerce!!**) behave similarly.

Note that when ***set** is used to copy values from a source pvar into a destination pvar, the source pvar is copied and then type converted if necessary. The (possibly converted) copy of the source pvar is then stored in the destination pvar. No coercion takes place on the original copy of the source pvar.

(pvar boolean) **boolean–pvar**

Boolean pvars have no parameters associated with them and are therefore never mutable.

When boolean values are stored in a general pvar, no type conversion is performed.

Within ***set** forms, boolean destination pvars can receive 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)

Front-end pvars have no parameters associated with them and are therefore never mutable.

When front-end values are stored in a general pvar, no type conversion is performed.

Within ***set** forms, front-end destination pvars can receive data of type **front–end** 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) **string–char–pvar**

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 converted to type **character.**

Within ***set** forms, **string–char** destination pvars can receive data of type **string–char** or type **character** only. If the *source* pvar is of the **character** data type, then the expression **(*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 **(*and (string–char–p!!** *source***))** returns **t** for the general pvar *source*.

(pvar character) **character–pvar**

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*)) **(field-pvar *length*)**

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, contain 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*)) **(signed-pvar *length*)**

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. Mutable **signed-byte** pvars are initially allocated 2 bits per processor. They can, however, contain 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 of any bit length is allowed. Otherwise, it must be possible to represent every active datum in the source using the same number of bits as the **signed-byte** 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***))**

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 GSL***)** significand length and **(max** *EL GEL***)** exponent length. If there was originally no floating-point data in the general destination pvar, this has no effect; *GSL* 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, i.e., pvars that are declared to be of the type **(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 is a mutable **defined–float** pvar, 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 may be signaled depending on the interpreter or compiler safety level in use.

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 a mutable **defined–float** pvar, then a copy of both 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***)))**

*Lisp supports complex pvars with real and imaginary parts of type **defined–float** only.

The restrictions on complex pvar parameters are identical to the restrictions on **defined–float** pvar 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, this form defines a mutable complex pvar:

```
(*proclaim '(type (pvar (complex (defined-float * *)))
mcmplx))
```

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** data. 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 pvars of type **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***))**

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***)**

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 that 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.

Chapter 5

*Lisp Compiler Options

This chapter describes the many compiler options you can use to control the way in which your *Lisp code is compiled, and also describes the means by which you can modify those options.

5.1 Setting Compiler Options

The compiler options control the behavior of the *Lisp compiler, including the degree of optimization it performs while generating code. There are two ways to set the compiler options: using a menu and directly modifying the values of *Lisp global variables.

5.1.1 Using the Compiler Options Menu

The options menu can be displayed by typing:

```
> (in-package '*lisp)
> (compiler-options)
```

For The Curious: You can also display the current settings of the *Lisp compiler options (without modifying them) by typing:

```
(slc::report-options)
```

In the Lucid Common Lisp version of *Lisp this function takes an optional argument that if non-**nil** adds the Lucid compiler options to the displayed list:

```
(slc::report-options t)
```

5.1.2 The Standard Options Menu

The standard options menu lists the following options. (Default values are shown.)

```
Starlisp Compiler Options

Compile Expressions (Yes, or No) Yes
Warning Level (High, Normal, None) Normal
Inconsistency Reporting Action (Abort, Error, Cerror, Warn, None) Warn
Safety (0, 1, 2, 3) 1
Print Length for Messages (an integer, or Nil) 4
Print Level for Messages (an integer, or Nil) 3
Pull Out Common Address Expressions (Yes, or No) No
Use Always Instructions (Yes, or No) No
```

On a UNIX front end, options are listed one at a time, each with its current value. To keep the current value for an option and go on to the next option, press Return. To change the option, type the desired value and press Return. At the end of the options list, confirmation is requested:

```
Do the assignment? (Yes, or No)
```

To save the options you've selected, type Yes and press Return. To cancel the changes you've made, type No and press Return.

5.1.3 The Extended Compiler Options Menu

Not all available options for controlling the behavior of the *Lisp compiler are listed by default when the options menu is invoked. The options that are not in the default menu provide capabilities that are not generally needed.

To invoke the options menu with all options listed, type the following:

```
(compiler-options :class :all)
```

The extended options menu lists the following options. (Default values are shown.)

```
Starlisp Compiler Options

Compile Expressions (Yes, or No) Yes
Warning Level (High, Normal, None) Normal
Inconsistency Reporting Action (Abort, Error, Cerror, Warn, None) Warn
Safety (0, 1, 2, 3) 1
Print Length for Messages (an integer, or Nil) 4
Print Level for Messages (an integer, or Nil) 3
Optimize Bindings (No, Cspeed<3, Yes) Cspeed<3
Peephole Optimize Paris (No, Cspeed<3, Yes) Cspeed<3
Pull Out Common Address Expressions (Yes, or No) No
Use Always Instructions (Yes, or No) No
Machine Type (Current, Compatible, Cm1, Cm2, Cm2-FPA, Simulator) Current
Add Declares (Everywhere, Yes, No) No
Use Undocumented Paris (Yes, or No) Yes
Verify Type Declarations (No, Current-Safety, Yes) Current-Safety
Constant Fold Pvar Expressions (Yes, or No) Yes
Speed (0, 1, 2, 3) 1
Compilation Speed (0, 1, 2, 3) 1
Space (0, 1, 2, 3) 1
Strict THE Type (Yes, or No) Yes
Immediate Error If Location (Yes, or No) Yes
Optimize Check Stack Expression (Yes, or No) Yes
Generate Comments With Paris Code (Yes, Macro, No) Yes
```

Using the Compiler Menu on a Symbolics Front End

On a Symbolics front end, changes are made by clicking the mouse on desired options and by typing new values where appropriate. To exit the menu and save the options you've selected, click the left mouse button on the Exit box. To exit the menu without saving the new selections, click on the Abort box.

Also, there are two alternate methods of invoking the options menu on a Symbolics front end:

- At a Lisp Listener, type the command

  ```
  :Set Compiler Options
  ```

- In the editor, type

  ```
  meta-x Set Compiler Options
  ```

5.1.4 Setting *Lisp Compiler Variables Directly

In addition to using the compiler options menu, compiler options may be changed by changing the value of associated *Lisp global variables, or, for certain options, by using a global declaration.

To set the values of compiler option variables, use the following operators:

setq **compiler–let** **optimize/*optimize**

These operators are described below, along with examples of their use.

setq

The simplest way to interactively modify the value of a compiler variable is to **setq** it to a new value. For example, you'll often want to modify the values of the compiler variables ***warning–level*** and ***safety***. You can use **setq** to change them, like this:

```
(setq *warning-level* :high *safety* 3)
```

compiler–let

The Common Lisp special form **compiler–let** can be used to selectively change the value of any *Lisp compiler option for a region of code. For example

```
(compiler-let ((*compilep* t) (*safety* 0)
               (*use-always-instructions* t))
    ...)
```

insures that the *Lisp compiler operates with a safety level of 0 and enables the use of Paris **–always** instuctions for the region of code enclosed by the **compiler–let** form.

optimize
*optimize

The Common Lisp **optimize** declaration specifier may be used within either a ***proclaim** form or a **declare** form to change optimization levels for both the Common Lisp compiler and *Lisp compiler. The ***optimize** declaration specifier, used within a *proclaim or a **declare** form, changes the optimization level for the *Lisp compiler only; it does not affect the Common Lisp compiler.

The following properties may be set by using **optimize** and ***optimize**:

safety **speed** **space** **compilation–speed**

For example,

```
(*proclaim '(optimize (safety 3)))
```

sets the **safety** level to 3 for both compilers, and

```
(*proclaim '(*optimize (safety 3)))
```

sets the **safety** level to 3 only for the *Lisp compiler.

The Common Lisp **declare** form may be also used with either the **optimize** or the ***optimize** declaration specifier to change the *Lisp optimization levels. For example:

```
(*let ((truth t!!))
    (declare (optimize (safety 3)))
    (foo (bar truth)))
```

In this example the **declare** form sets both the Common Lisp and the *Lisp safety levels at 3 for the entire body of the ***let** form.

5.2 *Lisp Compiler Options

All compiler options are listed below, in alphabetical order. Each is listed in the form

Name

Values: *legal values for this option*
Default: *the default value for this option*
Variable: *the global variable associated with this option*
A description of the compiler option, and of the effects of each of its values.

Note: Often the value displayed for a compiler option on the options menu will not be the same as the corresponding Lisp value stored in the compiler variable. For example, many compiler options are displayed as Yes or No choices on the menu, yet the corresponding variable will have values of either **t** or **nil**. In such cases, the appropriate Lisp values for the compiler option will be shown in parentheses after the values that appear on the options menu.

Add Declares

Values: Everywhere (**:everywhere**), Yes (**t**), No (**nil**)
Default: No (**nil**) on Symbolics front ends, Yes (**t**) on other front ends
Variable: ***add-declares***

The **Add Declares** compiler option determines if and how the *Lisp compiler will generate code that includes type declarations for stack address computations.

A value of Everywhere (**:everywhere**) causes the compiler to generate type declarations using both **declare** and **the** forms. A **the** form is used wherever **declare** is not legal.

A value of Yes (**t**) causes the compiler to generate type declarations wherever a **declare** form is appropriate.

A value of No (**nil**) prevents the compiler from generating any type declarations. The default value on Symbolics front ends is **nil** because the Symbolics implementation generally ignores type declarations.

Compile Expressions

Values: Yes (**t**), No (**nil**)
Default: Yes (**t**)
Variable: ***compilep***

The **Compile Expressions** option enables or disables the *Lisp compiler.

A value of Yes (**t**) enables the *Lisp compiler; a value of No (**nil**) disables it.

By default, the compiler is enabled.

Compilation Speed

Values: 0, 1, 2, 3
Default: 1
Variable: ***compilation-speed***

Note: Except as a constraint on the **Optimize Bindings** and **Peephole Optimize Paris** options, the **Compilation Speed** option is not currently used by the *Lisp compiler.

The **Compilation Speed** compiler option advises both the Common Lisp and the *Lisp compilers of the relative importance of compilation speed.

A value of 0, (low compilation speed) means compilation speed is totally unimportant.

A value of 1, the default, means compilation speed is of little importance.

A value of 2 means compilation speed is of moderate importance.

A value of 3 means compilation speed is extremely important. **Note:** At this value, both **Optimize Bindings** and **Peephole Optimize Paris** are disabled.

Constant Fold Pvar Expressions

Values: Yes (**t**), No (**nil**)
Default: Yes (**t**)
Variable: ***constant-fold***

The **Constant Fold Pvar Expressions** compiler option determines whether or not the *Lisp compiler will constant fold certain pvar expressions.

A value of Yes (**t**) allows the compiler to constant fold pvar expressions in which all arguments to certain *Lisp functions contain identical values in all active processors. Examples of these kinds of arguments are **nil!!**, **t!!**, and calls to the function **!!** (this includes scalar constants that are promoted to pvars).

A value of No (**nil**) prevents the compiler from constant folding.

For example, with this option enabled, expressions containing constant arguments, such as:

```
(+!! (the (unsigned-byte 32) x-position) 128 32 5)
```

are automatically simplified by performing the obvious arithmetic on the front-end. For example, the above expression is simplified to:

```
(!! (the (unsigned-byte 32) (+ x-position 128 32 5)))
```

Constant-folding is done wherever possible. For example, the expression

```
(+!! (the (unsigned-byte-pvar 32) x-position) 128 32 5)
```

is simplified to

```
(+!! (the (unsigned-byte-pvar 32) x-position) 165)
```

Constant folding can often make *Lisp code more efficient.

For example, with constant folding enabled,

```
(*sum (-!! 1.0))
```

compiles into:

```
(progn  ;; Constant global sum - *sum.
   (* -1.0  (cm:global-count-always cm:context-flag)))
```

whereas without constant folding, the same expression compiles into:

```
(let* ((slc::old-next-stack-field (cmi::next-stack-field))
       (-!!-index-2 (+ slc::old-next-stack-field 32)))
 (prog1
 (progn
    (cm:allocate-stack-field
         (- -!!-index-2 slc::old-next-stack-field))
    ;; Move constant - !!.
    (cm:move-constant slc::old-next-stack-field 1065353216 32)
    (cm:lognot(+ slc::old-next-stack-field 31)
                         (+ slc::old-next-stack-field 31) 1)
    (cmi::global-float-add slc::old-next-stack-field 23 8)
    (cm:deallocate-upto-stack-field slc::old-next-stack-
field)))
```

Clearly, constant folding allows the compiler to generate more efficient code.

Generate Comments With Paris Code

Values: Yes (**t**), Macro (**:macro**), No (**nil**)
Default: Yes (**t**)
Variable: ***generate–comments***

The **Generate Comments With Paris Code** compiler option controls whether or not the *Lisp compiler inserts comments into the Lisp/Paris code it generates.

A value of Yes (**t**) causes the compiler to generate comments

A value of Macro (**:macro**) causes the compiler to generate comments when forms are macroexpanded using the Symbolics editor command Macro Expand Expression.

A value of No (**nil**) prevents the compiler from placing comments in Lisp/Paris code.

Immediate Error If Location

Values: Yes (**t**), No (**nil**)
Default: Yes (**t**)
Variable: ***immediate–error–if–location***

The **Immediate Error If Location** option may be changed at run time to change the level of safety used by code compiled at a **Safety** level of 2.

The default value of Yes (**t**) makes such code run as if compiled at Safety level 3.

A value of No (**nil**) makes the code run as if compiled at Safety level 1.

See the description of the **Safety** compiler option for more information.

Inconsistency Reporting Action

Values: Abort (**:abort**), Error (**:error**), Cerror(**:cerror**),
Warn (**:warn**), None (**:none**)
Default: Warn (**:warn**)
Variable: ***inconsistency–action***

The **Inconsistency Reporting Action** option controls the behavior of the compiler when an inconsistency is discovered. An inconsistency usually indicates an implementation error in the compiler.

An value of Abort (**:abort**) causes the compiler to report a discovered compiler inconsistency and immediately abort the compilation.

A value of Error (**:error**) causes the compiler to report a discovered compiler inconsistency using the Common Lisp function **error**. This signals a fatal error and enters the debugger.

A value of Cerror (**:cerror**) causes the compiler to report a discovered compiler inconsistency using the Common Lisp function **cerror**. This signals a continuable error and enters the debugger. The program may be resumed after the error is resolved.

The default value of Warn (**:warn**) causes the compiler to report a discovered compiler inconsistency using the Common Lisp function **warn**. This prints a warning message but normally does not enter the debugger.

A value of None (**:none**) instructs the compiler not to take any special action when an inconsistency in the compiler is discovered.

Machine Type

Values: Current (**:current**), Compatible (**:compatible**), CM1 (**:cm1**), CM2 (**:cm2**), CM2–FPA (**:cm2–fpa**), Simulator (**:simulator**)
Default: Current (**:current**)
Variable: ***machine–type***

Note: This option is not currently used by the *Lisp compiler.

The **Machine Type** option directs the *Lisp compiler to generate code that is either specific to one of the Connection Machine models or compatible across models.

The default value of Current (**:current**) instructs the compiler to generate code specific to the current machine type.

A value of Compatible (**:compatible**) instructs the compiler to generate code compatible across machine types.

A value of CM1 (**:cm1**) allows the compiler to generate code specific to Connection Machine model CM–1.

A value of CM2 (**:cm2**) allows the compiler to generate code specific to the CM-2.

A value of CM2–FPA (**:cm2–fpa**) allows generation of code specific to the CM-2 with the floating-point accelerator. When machine type CM2–FPA is specified, the *Lisp compiler generates Paris instructions that take advantage of the floating point accelerator hardware. This is the most useful value of the **Machine Type** option.

A value of Simulator (**:simulator**) allows the compiler to generate code specific to the simulator. **Note:** This value is currently equivalent to the Compatible setting.

The example below demonstrates how the **Machine Type** option interacts with other compiler options. Code generated by compiling a ***sum** expression using three different combinations of the **Machine Type** and **Use Always Instructions** options is shown. Each successive combination produces more efficient code. Safety is set to 0 in all cases to eliminate error detection code, so that the examples are more readable.

Consider the following *Lisp code:

```
(*proclaim '(type (pvar single-float) sf1 sf2))
(*sum (*!! (+!! sf1 (!! 128.0)) sf2))
```

When the **Machine Type** option is set to Compatible (**:compatible**) and the **Use Always Instructions** option is set to No (**nil**), the compiler generates the following code:

```
(let* ((slc::old-next-stack-field (cm:allocate-stack-field 32))
         (*!!-index-2 (+ slc::old-next-stack-field 32)))
 (declare (ignore *!!-index-2))
 (prog1
  (progn ;; Move constant - !!.
    (cm:move-constant slc::old-next-stack-field 1124073472 32)
    (cmi::clear-mem cm:overflow-flag)
    (cm:f-add-2-11 slc::old-next-stack-field
         (pvar-location sf1) 23 8)
    ;; The result of a (two argument) float +!! overflowed.
    (cmi::error-if-location cm:overflow-flag 394259 nil)
    (cm:f-multiply-2-11 slc::old-next-stack-field
         (pvar-location sf2) 23 8)
    ;; The result of a (two argument) float *!! overflowed.
    (cmi::error-if-location cm:overflow-flag 394003)
    (cmi::global-float-add slc::old-next-stack-field 23 8))
    (cm:deallocate-upto-stack-field slc::old-next-stack-
field)))
```

However, when **Machine Type** is set to CM2–FPA (**:cm2-fpa**) and **Use Always Instructions** is set to No (**nil**), the compiler generates the following, more efficient, code:

```
(let* ((slc::old-next-stack-field (cm:allocate-stack-field 32))
        (*!!-index-2 (+ slc::old-next-stack-field 32)))
 (declare (ignore *!!-index-2))
 (prog1
  (progn
    (cmi::clear-mem cm:overflow-flag)
    (cm:f-add-constant-3-11 slc::old-next-stack-field
         (pvar-location sf1) 128.0 23 8)
    ;; The result of a (two argument) float +!! overflowed.
    (cmi::error-if-location cm:overflow-flag 394259 nil)
    (cm:f-multiply-2-11 slc::old-next-stack-field
         (pvar-location sf2) 23 8)
    ;; The result of a (two argument) float *!! overflowed.
    (cmi::error-if-location cm:overflow-flag 394003)
    (cmi::global-float-add slc::old-next-stack-field 23 8))
    (cm:deallocate-upto-stack-field slc::old-next-stack-
field)))
```

The most efficient code is generated when **Machine Type** is set to CM2–FPA (**:cm2–fpa**) and **Use Always Instructions** is set to Yes (**t**):

```
(let* ((slc::old-next-stack-field (cm:allocate-stack-field 32))
         (*!!-index-2 (+ slc::old-next-stack-field 32)))
 (declare (ignore *!!-index-2))
 (prog1
  (progn
    (cmi::clear-mem cm:overflow-flag)
    (cm:f-add-const-always-3-11 slc::old-next-stack-field
         (pvar-location sf1) 128.0 23 8)
    ;; The result of a (two argument) float +!! overflowed.
    (cmi::error-if-location cm:overflow-flag 394259 nil)
    (cm:f-multiply-always-2-11 slc::old-next-stack-field
         (pvar-location sf2) 23 8)
    ;; The result of a (two argument) float *!! overflowed.
    (cmi::error-if-location cm:overflow-flag 394003)
    (cmi::global-float-add slc::old-next-stack-field 23 8))
    (cm:deallocate-upto-stack-field slc::old-next-stack-
field)))
```

Macroexpand Inline Forms

Note: this option applies only to users on Symbolics front ends.

Values: Yes (**t**), No (**nil**)
Default: Yes (**t**)
Variable: ***macroexpand–inline–forms***

This option controls the way the command **Macro Expand Expression All** expands inline function forms.

The default value of **t** causes the command **Macro Expand Expression All** to expand inline forms as if they were macros.

A value of **nil** prevents the command **Macro Expand Expression All** from expanding inline forms as if they were macros.

Expanding inline function forms as if they were macros may make the *Lisp compiler's output more difficult to read. For example, consider the following ***set** expression:

```
(*set u8 u4)
```

With **Macroexpand Inline Forms** set to **nil**, an invocation of **Macro Expand Expression All** displays:

```
(progn
    ;; Move (coerce) source to destination - *set.
    (cm:unsigned-new-size (pvar-location u8)
                                          (pvar-location u4) 8 4)
    nil)
```

With **Macroexpand Inline Forms** set to **t**, an invocation of **Macro Expand Expression All** displays:

```
(progn
    ;; Move (coerce) source to destination - *set.
    (cm:unsigned-new-size (aref u8 1)
                                          (aref u4 1) 8 4)
    nil)
```

Notice that function calls like **pvar–location** have been turned into calls to **aref.**

Macroexpand Print Case

Note: this option applies only to users on Symbolics front ends.

Values:	No (**nil**), Downcase (**:downcase**), Upcase (**:upcase**) Capitalize (**:capitalize**)
Default:	No (**nil**)
Variable:	***macroexpand–print–case***

This option controls the print case used to display the expansions produced by the **Macroexpand Expression** command.

A **Macroexpand Expression** value of **nil** (the fault) causes the value of the variable ***print–case*** to be used.

A non–**nil** **Macroexpand Expression** value is used instead of ***print–case***.

Macroexpand Repeat

Values: Yes (**t**), No (**nil**)
Default: Yes (**t**)
Variable: ***macroexpand–repeat***

Note: this option applies only to users on Symbolics front ends.

This option controls the way the command **Macro Expand Expression** works.

A value of **t** causes **Macro Expand Expression** to use the Common Lisp **macroexpand** function, which repeatedly calls **macroexpand–1** to expand a macro expression.

A value of **nil** causes **Macro Expand Expression** to use the Common Lisp **macroexpand–1** function, which does not repeat.

Optimize Bindings

Values: No (**nil**), Cspeed<3 (**cspeed<3**), Yes (**t**)
Default: Cspeed<3 (**cspeed<3**)
Variable: ***optimize–bindings***

The **Optimize Bindings** option provides control over compilation speed by altering the number of temporary bindings generated by the *Lisp compiler.

A value of Yes (**t**) enables this option and causes extra bindings to be removed. When binding optimization is enabled, some temporary variables are eliminated and others are used repeatedly.

A value of No (**nil**) disables binding optimization. When the binding optimization option is disabled, the code produced by the compiler is more readable because it uses unique temporary address variables to represent each value represented.

The default value of Cspeed<3 varies binding optimization based on the value of the ***compilation–speed*** variable. If compilation speed is 3 (the highest possible value), then ***optimize–bindings*** is set to **nil**. If compilation speed is less than 3, then ***optimize–bindings*** is set to **t**.

Optimize Check Stack Expression

Values: Yes (**t**), No (**nil**)
Default: Yes (**yes**)
Variable: ***optimize–check–stack***

The **Optimize Check Stack Expression** compiler option determines how the *Lisp compiler manages the temporary stack space used by the Lisp/Paris code it generates.

The default value of Yes (**t**) makes the compiler try to remove the length expression from calls to **cm:allocate–stack–field.**

A value of No (**nil**) disables this optimization.

Peephole Optimize Paris

Values: No (**nil**), Cspeed<3 (**3**), Yes (t)
Default: Cspeed<3 (**3**)
Variable: ***optimize–peephole***

The **Peephole Optimize Paris** option controls the *Lisp compiler's peephole optimization of generated Lisp/Paris code.

A value of Yes (**t**) causes the *Lisp compiler to optimize the Lisp/Paris code it generates. A value of No (**nil**) prevents this optimization.

The default value of Cspeed<3 varies peephole optimization based on the value of the ***compilation–speed*** variable. If compilation speed is 3 (the highest possible value), then ***optimize–peephole*** is set to **nil**. If compilation speed is less than 3, then ***optimize–peephole*** is set to **t**.

Print Length for Messages
Print Level for Messages

Values: an integer or **nil**
Length Default: 4
Level Default: 3
Variables: ***slc–print–length*** ***slc–print–level***

These options control how much of a list expression the compiler prints when generating a warning about that expression.

As in Common Lisp, the Print Level indicates how many levels of data object nesting will be printed, counting from 0.

The Print Length indicates how many elements at each level will be printed, counting from 1.

For both variables, if the value **nil** is specified, no limit is imposed.

The Common Lisp variables ***print–length*** and ***print–level*** are bound to these variables when compiler messages are printed.

Pull Out Common Address Expressions

Values: Yes (**t**), No (**nil**)
Default: No (**t**)
Variable: ***pull–out–subexpressions***

Note: This option is not fully implemented and therefore may not work in some cases.

The **Pull Out Common Address Expressions** option determines whether the compiler performs common subexpression elimination on address expressions such as calls to **pvar–location**. Enabling this option can, in certain circumstances, increase performance significantly.

A value of Yes (**t**) enables this optimization; a value of No (**nil**) disables it. This optimization is off by default.

When enabled, this option trims the code executed on the front end; it does not affect the code executed on the Connection Machine. If a program already has a high Connection Machine utilization, this option will do little to improve the execution time. Conversely, if a program has a low Connection Machine utilization, enabling **Pull Out Common Address Expressions** can reduce execution time. The potential benefit is usually greater for larger expressions, where there are more opportunities for common addressing expressions.

For example, consider the following ***set** expression:

```
(*set s16 (+!! (*!! s8 s8-2) s16-2))
```

Here is the code produced with this option *disabled*:

```
(progn
    (cm:multiply   (pvar-location s16) (pvar-location s8)
                        (pvar-location s8-2) 16 8 8)
    (cm:+ (pvar-location s16) (pvar-location s16-2) 16)
    (cmi::error-if-location cm:overflow-flag 66575)
   nil)
```

Here is the code produced by the compiler with this option *enabled*:

```
(let* ((pvar-location-s16-1 (pvar-location s16))
         (pvar-location-s8-2 (pvar-location s8))
         (pvar-location-s8-2-3 (pvar-location s8-2))
         (pvar-location-s16-2-4 (pvar-location s16-2)))
   (cm:multiply    pvar-location-s16-1 pvar-location-s8-2
                       pvar-location-s8-2-3     16 8 8)
   (cm:+      pvar-location-s16-1 pvar-location-s16-2-4 16)
   (cmi::error-if-location cm:overflow-flag 66575)
   nil)
```

Notice that **pvar–location** is executed four times when **Pull Out Common Address Expressions** is enabled, versus five times when it is disabled.

Rewrite Arithmetic Expressions

Values: Yes (**t**), No (**nil**)
Default: Yes (**t**)
Variable: ***rewrite–arithmetic–expressions***

This option determines whether the compiler optimizes arithmetic operations such as

```
(*set x (+!! x y z))
```

using the associative rules of arithmetic.

The default value of Yes (**t**) allows the compiler to rewrite arithmetic operations as if they were associative.

A value of No (**nil**) prevents this arithmetic-rewriting optimization.

When this option is enabled, the *Lisp compiler may produce more efficient code in some cases.

When this option is disabled, the *Lisp compiler evaluates expressions in the order in which they appear.

Regardless of the current **Rewrite Arithmetic Expressions** setting, you can force a specific order of evaluation by explicitly directing the computation:

```
(progn (*set x (+!! x y)) (*set x (+!! x z)))
```

Usage Note: When computing with floating-point data, results may vary depending on how this option is set. For example, consider the expression

```
(*set x (+!! x y z))
```

The laws of arithmetic allow this to be computed as either of the following expressions:

```
(*set x (+!! x (+!! y z)))    (*set x (+!! (+!! x y) z))
```

Given the limitations imposed by fixed-precision floating-point arithmetic, the two ways of evaluating the original expression may not yield identical results if x, y, and z are floating-point or complex pvars.

Safety

Values: 0, 1, 2, 3
Default: 1
Variable: ***safety***

The **Safety** option controls what kind of code the compiler generates to detect error conditions, and also controls how these error conditions are reported.

At a safety level of 0 (low safety) no error-checking code is generated.

At the default safety level of 1, limited error-checking code is generated, so an error may not be signalled at the exact point in your code at which it occurred.

At a safety level of 2, the generated code implements either level 1 or level 3 safety, depending on the value of the compiler variable ***immediate–error–if–location***. (See description of the **Immediate Error If Location** compiler option.)

At a safety level of 3, (high safety), full error-checking code is generated, so that an error will always be signalled at the exact point in your code at which it occurred.

In general, high safety produces slow but safe code, and should be used for debugging purposes, while low safety produces the fastest code.

Space

Values: 0, 1, 2, 3
Default: 1
Variable: ***space***

Note: This option is not currently used by the *Lisp compiler.

The **Space** compiler option advises both the Common Lisp and the *Lisp compilers of the relative importance of the space utilization of compiled code, including both the size of the generated code and its run-time space utilization.

A value of 0, means code size and instruction space utilization are totally unimportant.

A value of 1, the default, means code size and space utilization are of little importance.

A value of 2 means code size and space utilization are of moderate importance.

A value of 3 means code size and space utilization are extremely important.

Speed

Values: 0, 1, 2, 3
Default: 1
Variable: ***speed***

Note: This option is not currently used by the *Lisp compiler.

The **Speed** compiler option advises both the Common Lisp and the *Lisp compilers of the relative importance of speed in the resulting code.

A value of 0, (low speed) means speed of execution is totally unimportant.

A value of 1, the default, means speed of execution is of little importance.

A value of 2 means speed of execution is of moderate importance.

A value of 3 means speed of execution is extremely important.

Use Always Instructions

Values: Yes (**t**), No (**nil**)
Default: No (**nil**)
Variable: ***use–always–instructions***

Note: This option may generate undocumented Paris instructions.

The **Use Always Instructions** option determines whether or not the *Lisp compiler generates unconditional **–always** Paris instructions for stack operations.

A value of Yes (**t**) enables the use of the Paris **–always** instructions; a value of No (**nil**) disables their use. This option is disabled by default.

For an example of code generated when this option is set to Yes, see the last example under the **Machine Type** option description.

Use Code Walker

Values: Yes (**t**), No (**nil**)
Default: Yes (**t**)
Variable: **slc::*use–code–walker***

This option controls whether the code walker portion of the *Lisp compiler is enabled.

The default value of Yes (**t**) enables the code walker. A value of No (**nil**) disables the code walker.

The code walker allows the *Lisp compiler to find type declarations it would otherwise miss, and to compile *Lisp code more thoroughly.

If the code walker is enabled, the compiler sees declarations in all locations permitted by Common Lisp, and will compile all properly declared code.

If the code walker is disabled, the compiler will only see declarations within ***defun**, ***let**, ***let***, and ***locally** forms, and will only compile code within these *Lisp forms:

***set**	***pset**	***setf**	**pref**	***sum**	***integer–length**
***or**	***and**	***xor**	***logior**	***logand**	***logxor**
***max**	***min**	***locally**			

Additionally, the predicates for ***when**, ***unless**, ***if**, and ***cond** and the variable initialization forms for ***let** and ***let*** variables will be compiled, but the body code of these forms will not.

Use Undocumented Paris

Values: Yes (**t**), No (**nil**)
Default: Yes (**t**)
Variable: ***use–undocumented–paris***

The **Use Undocumented Paris** compiler option determines whether or not the code generated by the *Lisp compiler uses undocumented Paris instructions.

The default value of Yes (**t**) allows the use of undocumented Paris instructions. In many cases, enabling this option significantly increases the execution speed of compiled *Lisp code.

A value of No (**nil**) disallows the use of most undocumented Paris instructions.

For example, with **Use Undocumented Paris** set to Yes (**t**), compiling

```
(*sum (if!! b1 s8 s8-2))
```

results in code that includes three internal, undocumented Paris functions in the **CMI** package. When the same ***sum** statement is compiled with this option set to No (**nil**), the generated code includes only documented functions in the **CM** package.

If the **Use Undocumented Paris** option is disabled, it still allows the *Lisp compiler to generate undocumented Paris routines in cases where no appropriate documented Paris instructions exists. However, if a documented instruction exists, it will be used, even if the undocumented instruction is faster.

Verify Type Declarations

Values: No (**nil**), Current–Safety (**:current–safety**), Yes (**t**)
or an integer between 0 and 3
Default: Current–Safety (**:current–safety**)
Variable: ***verify–type–declarations***

The **Verify Type Declaration** compiler option determines whether or not the *Lisp compiler generates type verification code for arguments to user-defined functions that have been given either **the** or **declare** type declarations.

This option is primarily useful for debugging *Lisp programs. The most common user errors are declaring pvar arguments incorrectly and violating type declarations.

These errors are often hard to track down because the results of violating a type declaration can be unpredictable. With the **Safety** option set at 3, and the **Verify Type Declarations** option enabled, the compiler generates code to catch erroneous and violated type declarations immediately.

The legal integer values for this option are:

0 No error checking is done.
1 Minimal error checking is done.
2 Moderate error checking is done (more than level 1, but less than level 3).
3 Full type verification error checking is done.

A value of Yes (**t**) causes to the compiler to generate the maximum amount of error checking code, and is equivalent to a value of 3.

A value of No (**nil**) prevents the compiler from generating any type verification code and is equivalent to a value of 0.

The default value of Current–Safety (**:current–safety**) sets the verification level based on the current safety level. If the **Safety** option is set to 0, and **Verify Type Declarations** is set to Current–Safety, no verification code is generated. With **Safety** at **3**, verification becomes likewise set to **3**, and so on.

As an example, consider the following ***sum** expression.

```
(*sum (the (field-pvar 32) quux)))
```

At a **Verify Type Declarations** level of 0. the compiler generates no type checking code, so this ***sum** expression compiles into

```
(cm:global-unsigned-add (pvar-location quux) 32)
```

At **Verify Type Declarations** level 1, the compiler generates minor error checking code:

```
(progn
 (if (not (*lisp-i:internal-pvarp quux))
  (slc::error-doesnt-match-declaration
    quux '(pvar (unsigned-byte 32))))
 (cm:global-unsigned-add (pvar-location quux) 32))
```

In this case, a test is done to make sure that **quux** is a pvar.

At **Verify Type Declarations** level 2, the compiler generates more error checking code:

```
(progn
 (if (not (and(*lisp-i:internal-pvarp quux)
                       (eq (pvar-type quux) :field)))
  (slc::error-doesnt-match-declaration
    quux '(pvar (unsigned-byte 32))))
 (cm:global-unsigned-add (pvar-location quux) 32))
```

Here, the verification code insures that **quux** is a **field–pvar**.

At **Verify Type Declarations** level 3, the compiler generates the maximum error checking code:

```
(progn
 (if (not (and(*lisp-i:internal-pvarp quux)
                       (eq (pvar-type quux) :field)
                       (eql (pvar-length quux) 32)))
  (slc::error-doesnt-match-declaration
    quux '(pvar (unsigned-byte 32))))
 (cm:global-unsigned-add (pvar-location quux) 32))
```

In this case, the verification code tests that **quux** is a **field–pvar** of length 32.

Warning Level

Values: High (**:high**), Normal (**:normal**), None (**:none**)
Default: Normal (**:normal**)
Variable: ***warning–level***

The **Warning Level** option controls the warnings produced by the *Lisp compiler.

A warning level value of High (**:high**) causes the compiler to generate a warning whenever an expression is not compiled. The warning tries to explain why the expression is not compiled. Usually the cause is a lack of type declarations, as shown in the following example:

```
(*proclaim '(type (pvar (signed-byte 8)) s8))
(*set s8 (+!! s8 variable))
```

Attempting to compile the above code with the warning level set to High (**:high**), produces the following warning:

```
;;; Warning: *Lisp Compiler: While compiling VARIABLE:
;;; The expression (*LISP-I::*SET-1 S8 (+!! S8 VARIABLE)) is not compiled
;;; because the *Lisp compiler cannot find a declaration for VARIABLE
```

By contrast, the following form can be successfully compiled because the data type of **variable** is supplied.

```
(*proclaim '(type (pvar (signed-byte 8)) s8))
(*set s8 (+!! s8 (the (pvar (signed-byte 8)) variable)))
```

The default warning level of Normal (**:normal**) causes the compiler to generate warnings only for invalid function arguments and type mismatches.

For example, with warning level set to Normal (**:normal**), an attempt to compile

```
(*proclaim '(type (field-pvar 8) u8))
(*proclaim '(type boolean-pvar b1))
(*set u8 (-!! b1))
```

results in this warning:

```
Warning: While compiling B1:
Function -!! expected a numeric pvar argument but got a boolean pvar argument.
```

At a warning level value of None (**:none**) the compiler does not signal warnings.

Part II
*Lisp Dictionary

abs!! [Function]

Takes the absolute value of the supplied pvar.

SYNTAX

abs!! *numeric–pvar*

ARGUMENTS

numeric–pvar Numeric pvar. Pvar for which absolute value is calculated.

RETURNED VALUE

absolute–value–pvar
Temporary numeric pvar. In each active processor, contains the absolute value of the corresponding value of *numeric–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The **abs!!** function takes the absolute value of *numeric–pvar*. It returns a temporary pvar that contains in each active processor the absolute value of the corresponding value of *numeric–pvar*. The **abs!!** function provides the same functionality for numeric pvars as the Common Lisp function **abs** provides for numeric scalars.

EXAMPLES

For non-complex numeric pvars, **abs!!** returns the positive magnitude of *numeric–pvar* in each active processor. For example, the following are equivalent:

```
(abs!! pvar)      <=>  (if!! (minusp!! pvar) (-!! pvar) pvar)
(abs!! (!! -5))  <=>  (!! 5)
```

For complex pvars, **abs!!** returns the complex magnitude of *numeric–pvar* in each active processor, as a floating-point number.

```
(abs!! complex-pvar)  <=>
(sqrt!! (+!!   (expt!! (realpart!! complex-pvar) (!! 2))
               (expt!! (imagpart!! complex-pvar) (!! 2))))

(abs!! (!! #c(4 3)))  <=>  (!! 5.0)
```

NOTES

It is an error if any of the *numeric–pvar* arguments contains a non-numeric value in any active processor.

acos!!, acosh!! *[Function]*

Take the arc cosine and arc hyperbolic cosine of the supplied pvar.

SYNTAX

acos!! *numeric–pvar*
acosh!! *numeric–pvar*

ARGUMENTS

numeric–pvar Numeric pvar. Pvar for which the arc cosine (arc hyperbolic cosine) is calculated.

RETURNED VALUE

arc–cosine–pvar Temporary numeric pvar. In each active processor, contains the arc cosine (arc hyperbolic cosine) in radians of the corresponding value of *numeric–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The **acos!!** function calculates the arc cosine of *numeric–pvar* in all active processors. It returns a temporary pvar containing in each active processor the arc cosine in radians of the corresponding value of *numeric–pvar*. Similarly, the **acosh!!** function calculates the arc hyperbolic cosine of *numeric–pvar* in all active processors. The **acos!!** and **acosh!!** functions provide the same functionality for numeric pvars as the Common Lisp functions **acos** and **acosh** provide for numeric scalars.

EXAMPLES

If *numeric–pvar* contains non-complex values, **acos!!** returns the arc cosine in each active processor, while **acosh!!** returns the arc hyperbolic cosine in each active processor. For example:

```
(acos!! (!! -1.0))           <=>  (!! 3.1415927)
(acosh!! (!! 11.591953))     <=>  (!! 3.1415927)
```

If *numeric–pvar* contains complex values, **acos!!** returns the complex arc cosine in each active processor, while **acosh!!** returns the complex arc hyperbolic cosine in each active processor:

```
(acos!! (!! #c(-1.0 0.0)))         <=> (!! #c(3.1415927 0.0))
(acosh!! (!! #c(11.591953 0.0)))   <=> (!! #c(3.1415927 0.0))
```

NOTES

It is an error if *numeric–pvar* contains integer or floating-point values of magnitude greater than 1.0 in any active processor. Complex values with magnitude greater than 1.0 are allowed.

It is an error if *numeric–pvar* contains a non-numeric value in any active processor.

add–initialization [*Function*]

Appends a *Lisp form to one or more initialization lists, which are evaluated before and after ***cold–boot** and ***warm–boot.**

SYNTAX

add–initialization *name–of–form form init–list–name*

ARGUMENTS

name–of–form	Character string. Name of initialization being added.
form	Any *Lisp form. Code to evaluate at initialization time.
init–list–name	Symbol or list of symbols. Initialization list(s) to which the code is to be added.

RETURNED VALUE

nil	Executed for side effect.

SIDE EFFECTS

The list or lists specified by *init–list–name* are modified by appending the initialization specified by *form*.

DESCRIPTION

The function **add–initialization** adds a named initialization form to one or more of the following *Lisp initialization lists:

- ***before–*cold–boot–initializations***
  *Lisp code evaluated immediately prior to any call to ***cold–boot**.

- ***after–*cold–boot–initializations***
  *Lisp code evaluated immediately after any call to ***cold–boot**.

- ***before–*warm–boot–initializations***
  *Lisp code evaluated immediately prior to any call to ***warm–boot**.

- ***after–*warm–boot–initializations***
  *Lisp code evaluated immediately after any call to ***warm–boot**.

The forms in these lists are evaluated in the order in which they were added to the initialization lists.

The argument *name–of–form* is a character string that names the *Lisp code being added to the specified list(s). The argument *form* may be any executable *Lisp form.

The *init–list–name* must be either one of the initialization list symbols above or a list of these symbols. In the latter case, the *form* is added to each initialization list named.

The function **delete–initialization** may be called with *name–of–form* to remove the initialization from the list(s).

EXAMPLES

The function **add–initialization** is the correct way to add an initialization form to any of the above lists. For example,

```
(add-initialization "Recompute Important Pvars"
   '(recompute-important-pvars *number-of-processors-limit*)
   '*after-*cold-boot-initializations*)
```

adds an initialization named **"Recompute Important Pvars"** to the list ***after–*cold–boot–initializations***, which calls a user-defined function named **recompute–important–pvars** with the current number of processors.

The same initialization can be added to more than one list. For example,

```
(add-initialization "Yell About Booting"
   '(format t "*Lisp has just been booted.")
   '(*after-*cold-boot-initializations*
     *after-*warm-boot-initializations*))
```

adds an initialization to both ***after–*cold–boot–initializations*** and ***after–*warm–boot–initializations***, which displays a warning message immediately after any call to ***cold–boot** or ***warm–boot**.

Because **add–initialization** is a function, the *form* and *init–list–name* arguments must be quoted if they are not meant to be evaluated during the call to **add–initialization**.

NOTES

Adding two forms with the same name to the same list is permissible only if the forms are the same according to the function **equal**; otherwise an error is signaled.

REFERENCES

See also the related operation **delete–initialization.**

See also the following Connection Machine initialization operators:

***cold–boot** ***warm–boot**

See also the character attribute initialization operator **initialize–character.**

address–nth, address–plus–nth, address–rank *[Function]*

These are the scalar couterparts of the functions **address–nth!!**, **address–plus–nth!!**, and **address–rank!!**

address–nth returns the coordinate of an address object along a specified dimension.
address–plus–nth increments the coordinate of an address object for a specified dimension.
address–rank returns the number of coordinates specified by an address object.

SYNTAX

address–nth	*address–object dimension*	=>	*coordinate*
address–plus–nth	*address–object increment dimension*	=>	*inc–addresss–obj*
address–rank	*address–obj*	=>	*rank*

ARGUMENTS

address–object	Front-end address object, as created by the function **grid**.
dimension	Integer. Zero-based number of the dimension to be returned or incremented (for **address–nth** and **address–plus–nth** only).
increment	Integer. Amount by which the specified *dimension* is to be incremented (for **address–plus–nth** only).

RETURNED VALUE

coordinate	Integer. The coordinate of *address–object* along the dimension specified by *dimension*.
inc–addresss–obj	Address object. Copy of *address–obj* with the coordinate specified by *dimension* incremented by *increment*.
rank	Integer. Number of coordinates in *address–obj*.

SIDE EFFECTS

None.

DESCRIPTION

The function **address–nth** returns the grid (NEWS) coordinate of *address–object* along the dimension specified by *dimension*. The argument *dimension* must be an integer between 0 and one less than the number of dimensions in *address–object*.

The function **address–plus–nth** increments the *n*th coordinate of *address–obj*, where *n* is the grid (NEWS) dimension specified by *dimension*.

The function **address–rank** returns the number of coordinates in *address–obj*.

EXAMPLES

```
(setq addr-obj (grid 12 3 0 29))

(address-nth addr-obj 0)  => 12
(address-nth addr-obj 3)  => 29

(address-plus-nth addr-obj 5 0)  <=>  (grid 17 3 0 29)

(address-rank addr-obj) => 4
```

REFERENCES

See also the related operations

address–nth!!		**address–plus–nth!!**	**address–rank!!**
grid	**grid!!**	**grid–relative!!**	**self!!**

address–nth!!, address–plus–nth!!, address–rank!! [*Function*]

These functions perform simple operations on address object pvars.

address–nth!! creates an address object pvar containing the specified coordinates.
address–nth!! returns a copy of an address object pvar with each of its values incremented along the specified dimensions.
address–rank!! returns a pvar containing the rank of each value of an address object pvar.

SYNTAX

address–nth!! *address–obj–pvar dimension–pvar* => *coordinate–pvar*
address–plus–nth!! *address–obj–pvar increment–pvar dimension–pvar* => *inc–address–pvar*
address–rank!! *address–obj–pvar* => *rank–pvar*

ARGUMENTS

address–obj–pvar Address object pvar, as created by the function **grid!!**.

dimension–pvar Integer pvar. Zero-based number of the dimension to be retrieved/incremented (**address–nth!!** and **address–plus–nth!!** only).

increment–pvar Integer pvar. Amount by which the coordinate specified by *dimension–pvar* is to be incremented (**address–plus–nth!!** only).

RETURNED VALUE

coordinate–pvar Temporary integer pvar. In each active processor, contains the coordinate of the corresponding value of *address–obj–pvar* along the dimension specified by *dimension–pvar.*

inc–address–pvar Temporary address object pvar. In each active processor, contains a copy of the value of *address–obj–pvar* with the coordinate specified by *dimension–pvar* incremented by *increment–pvar*.

rank–pvar Temporary integer pvar. In each processor, contains the number of coordinates in the corresponding value of *address–obj–pvar.*

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

For each processor, **address-nth!!** returns the *n*th grid (NEWS) coordinate of *address-object-pvar*, where *n* is the dimension specified by the corresponding value of *dimension-pvar*.

For each processor, **address-plus-nth!!** returns an address object pvar that is a copy of *address-obj-pvar* with the dimension specified by *dimension-pvar* incremented by *increment-pvar*.

For each processor, **address-rank!!** returns in each processor the number of coordinates in the corresponding value of *address-obj-pvar*.

EXAMPLES

```
(address-nth!! (grid!! x y z) (!! 1))  => (!! y)
(address-nth!! (grid!! x y z) (!! 2))  => (!! z)

(address-plus-nth!! (grid!! (!! x) (!! y) (!! z))
                    (!! 5) (!! 1))
  <=>
(grid!! (!! x) (+!! y (!! 5)) (!! z))

(address-rank!! (grid!! (!! x) (!! y)))  <=>  (!! 2)
```

REFERENCES

See also the related operations

address-nth		**address-plus-nth**	**address-rank**
grid	**grid!!**	**grid-relative!!**	**self!!**

alias!! [*Macro*]

Returns the actual contents of the specified subfield of a pvar, redefined as a temporary pvar of appropriate size and type.

SYNTAX

alias!! *subfield–selector*

ARGUMENTS

subfield–selector Pvar subfield selector. Must be a call to either **aref!!** or **row–major–aref!!**, a call to a structure pvar slot accessor defined by ***defstruct**, or a call to one of the functions **imagpart!!**, **realpart!!**, or **load–byte!!**.

RETURNED VALUE

aliased–pvar A temporary pvar of the same data type as the referenced pvar subfield, such that the data contained in the aliased pvar is identical to the data contained in the pvar subfield, rather than being a copy of the data (i.e., the aliased pvar references the same area of CM memory as the subfield selector.)

DESCRIPTION

In *Lisp, a parallel array accessor, such as **aref!!** or **row–major–aref!!**, returns a temporary pvar that is a copy of the element being referenced. Likewise, a parallel structure slot accessor, as defined by a call to ***defstruct**, returns a temporary pvar that is a copy of the parallel structure slot being accessed. Other pvar operations that return subfields of a pvar, such as **imagpart!!**, **realpart!!**, and **load–byte!!**, by definition return a copy of the referenced subfield. For most purposes, this copying is transparent and makes no difference.

Two important exceptions are:

- passing a pvar subfield to a user-defined function that must modify the subfield directly

- passing a pvar subfield to any function or macro where the size of the pvar subfield makes copying inefficient (i.e., a structure slot that contains another structure of considerable size).

In these two cases, the **alias!!** macro can be used to specify that the actual contents of the pvar subfield should be returned, rather than a copy.

The **alias!!** macro creates and returns a temporary pvar defined in such a way that the contents of the pvar are the actual contents of the referenced pvar subfield. The **alias!!** macro in effect "renames" or "aliases" the portion of a pvar referenced by the supplied *subfield–selector.* The *aliased–pvar* returned by **alias!!** may be freely referenced and modified as a pvar of the same data type as the pvar subfield.

Important: The **alias!!** macro is necessary only in the two cases mentioned above. In all other cases, use of the **alias!!** macro has no effect and detracts from readability of code. In some cases, explicit use of the **alias!!** macro is redundant. The following functions effectively perform an **alias!!** operation on their arguments:

***setf**	***pset**	***news**

EXAMPLES

The *subfield–selector* argument to **alias!!** can be an array reference, i.e., a call to either **aref!!** or **row–major–aref!!**. For example, given the array defined by

```
(*defvar array-pvar (!! #2A((1 2 3) (4 5 6))))
```

both of the following expressions modify the same element of the array.

```
(modify-array-element
   (alias!! (aref!! array-pvar (!! 1) (!! 1))))
(modify-array-element
   (alias!! (row-major-aref!! array-pvar (!! 4))))
```

The *subfield–selector* argument to **alias!!** can also be a structure slot reference, i.e., a call to a slot accessor function created by ***defstruct**.

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

```
(*defstruct history-struct
   (description nil :type (vector string-char 1000))
   (sickness-id 0 :type (unsigned-byte 32)))

(*defstruct patient
  (id-no 0 :type (unsigned-byte 8))
  (doctor 0 :type (unsigned-byte 8))
  (sick-p t : type boolean)
  (case-history nil :type (pvar (array history-struct (100))))
  )

(defun modify-patient-slot (slot-pvar value)
  (declare (type (field-pvar *) slot-pvar value))
  nil
  (*set slot-pvar value))

(defun in-error ()
  (*let ((ellen (make-patient!!)))
    (declare (type (pvar patient) ellen))
    (modify-patient-slot (patient-sick-p!! ellen) nil!!)
    (ppp (patient-sick-p!! ellen) :end 5)))

(defun correct ()
  (*let ((ellen (make-patient!!)))
    (declare (type (pvar patient) ellen))
    (modify-patient-slot
      (alias!! (patient-sick-p!! ellen)) nil!!)
    (ppp (patient-sick-p!! ellen) :end 5)))
```

The **in–error** function is in error because **(patient–sick–p!! ellen)** returns a temporary pvar containing a copy of the data in **ellen**'s **sick–p** slot. This pvar is allocated on the stack. The function **modify–foo–slot** then attempts to ***set** this temporary pvar, rather than the actual data stored in the structure **ellen**. The original data is not modified.

The **correct** function is correct because **alias!!** returns the actual slot **sick–p** from **ellen** as a pvar that can be modified by a call to the user-defined function **modify–patient–slot**.

The *subfield–selector* argument to **alias!!** can also be one of the pvar subfield operations **imagpart!!**, **realpart!!**, and **load–byte!!**. (Due to its implementation, **alias!!** cannot be applied to these three operators in the *Lisp simulator.)

For example,

```
(alias!! (imagpart!! complex-pvar))
(alias!! (realpart!! complex-pvar))
(alias!! (load-byte!! integer-pvar position-pvar size-pvar))
```

Besides passing pvar subfields to functions that modify those fields, **alias!!** may also be used to prevent copying of large pvar subfields.

For example, in the expression

```
(hypocondriac-p!! (alias!! (patient-case-history!! ellen)))
```

the user-defined function **hypochondriac-p!!** does *not* modify the **case-history** slot of **ellen**. Even so, using **alias!!** in this expression is more efficient because it prevents the possibly quite large **case-history** slot from being copied in the process of passing it to the function **hypochondriac-p!!**.

An example of when *not* to use the **alias!!** macro is provided by the expression

```
(*set dest-pvar
   (+!!  (alias!! (aref!! array-pvar (!! 0)))
         (alias!! (structure-slot!! structure-pvar))))
```

Neither of the calls to **alias!!** are necessary in this expression, because no modification of the referenced location takes place. It is also unnecessary and redundant to apply **alias!!** to the arguments of the *Lisp functions ***setf** and ***pset**. For example, in the expression

```
(*setf (alias!! (aref!! array-pvar (!! 3))) (!! 2))
```

the ***setf** macro effectively performs an **alias!!** operation on its first argument, so the extra call to **alias!!** is unnecessary.

Also, in many cases it is not necessary to use the operator **alias!!** in combination with **aref!!** to prevent the copying of large array pvars, because the *Lisp compiler is able to recognize and optimize cases where this copying is unnecessary. See the dictionary entry for **aref!!** for more information.

NOTES

The **alias!!** macro may not be applied to an array reference that uses indirect addressing, i.e., a call to **aref!!** with an index pvar containing different values in each processor. The **alias!!** macro also may not be applied to array accessors that operate on arrays in sideways (slicewise) orientation. These operators are:

sideways-aref!! **row-major-sideways-aref!!**

REFERENCES

See also the related operator **taken-as!**

*all [*Macro*]

Executes *Lisp forms with all processors selected.

SYNTAX

***all &body** *body*

ARGUMENTS

body	*Lisp forms. Any number of statements, which are executed in order.

RETURNED VALUE

body–value	Scalar or pvar value. Value of final form in *body.*

SIDE EFFECTS

Temporarily binds currently selected set to include all processors during execution of the forms in *body.*

DESCRIPTION

The macro ***all** is one of the processor selection operations. It executes a set of *Lisp forms with the currently selected set bound to include all processors in the current VP set. The value of the final expression in the body of the ***all** form is returned.

EXAMPLES

The most common use of the ***all** macro is to ensure that all processors are selected before the execution of a section of code. For example, the form

```
(*all (*set every-proc (!! 5)))
```

selects all processors and then uses ***set** to store 5 as the value of **every-proc** in every processor. Using ***all** guarantees that **every-proc** has the same value in every processor after this operation.

Processor selection macros can be nested. The expression

```
(*all
   (*set numeric-pvar (random!! (!! 10.0)))
   (*when (<!! numeric-pvar (!! 1))
      (*set numeric-pvar (/!! numeric-pvar))))
```

uses ***all** to select all processors, ***set** to store a random floating-point value between 0 and 10 into **numeric-pvar** for every processor, and ***when** to select only those processors in which the value stored in **numeric-pvar** is less than 1. In these processors, **/!!** is used to calculate the reciprocal of the value in **numeric-pvar**, and ***set** is used to store the calculated value back into **numeric-pvar**.

Because ***all** temporarily binds the currently selected set, and restores its original value upon exiting, it can be used within other processor selection macros to temporarily reselect all processors. For example, the expression

```
(*when (<!! data-pvar (!! 100))
   (/   (*sum data-pvar)
        (*all (*sum data-pvar))))
```

uses ***when** to select those processors in which the value of **data-pvar** is less than 100. The global function ***sum** is used to take the sum of the values in these processors. Then ***all** is used to temporarily rebind the currently selected set so that ***sum** can be used to take the sum of the values of **data-pvar** in all processors. The result returned by the entire expression is the ratio between the sum of the values of **data-pvar** that are less than 100 and the sum of all values of **data-pvar**.

NOTES

The ***cold-boot** and ***warm-boot** operations force reselection of all processors, but these operations also reset *Lisp and clear the *Lisp stack. See the definitions of ***cold-boot** and ***warm-boot** for more information.

It is not necessary to use ***all** around every body of code. The ***all** macro is only necessary only in three cases:

- Around the body of functions that need all processors active, but are called from within code that restricts the currently selected set.

- Around any code that requires all processors to be selected temporarily. For example, see the selective sum and division example above, which momentarily changes the currently selected set.

- Within code that changes the current VP set. Each VP set keeps track of its own currently selected set of active processors. To avoid using a previously restricted set of active processors when switching between VP sets, use ***all**.

An example of the last case is:

```
(def-vp-set fred '(16384))
(def-vp-set wilma '(8192))

(*with-vp-set fred
 (*when (<!! (self-address!!) (!! 100))
  (format t "~%In FRED, # active procs should be 100, ~
             and is: ~d" (*sum (!! 1)))
  (*with-vp-set wilma
   (format t "~%In WILMA, # active procs should be 8192, ~
             and is ~d" (*sum (!! 1)))
   (*with-vp-set fred
    (format t "~%In FRED, the # active procs should still ~
              be 100, and is ~d" (*sum (!! 1)))
    (*all
     (format t "In FRED, the # active procs should now ~
             be 16384, is ~D" (*sum (!! 1)))))
   (format t "~%In WILMA, # active procs should still ~
             be 8192, is: ~d" (*sum (!! 1))))
  (format t "~%In FRED, # active procs should again ~
             be 100, is: ~d" (*sum (!! 1)))))
```

This example produces the following output:

```
In FRED, # of active procs should be 100, and is: 100
In WILMA, # of active procs should be 8192, and is: 8192
In FRED, # of active procs should still be 100, and is: 100
In FRED, # of active procs should now be 16384, is: 16384
In WILMA, # of active procs should be 8192, is: 8192
In FRED, # of active procs should again be 100, is: 100
```

Note the use of ***all** within the ***with-vp-set** forms in this example to ensure that all the processors of the newly selected VP set are active. Note also the use of the *Lisp idiom **(*sum (!! 1))** to determine the number of active processors.

Forms such as **throw, return, return-from,** and **go** may be used to exit an external block or looping construct from within a processor selection operator. However, doing so will leave the currently selected set in the state it was in at the time the non-local exit form is executed. To avoid this, use the *Lisp macro **with-css-saved**. For example,

```
(defun safe-division (y x)
   (*when (evenp!! (self-address!!))
      (block division
        (with-css-saved
         (*all
            (*if (>!! y (!! 0))
                 (if (*or (=!! (!! 0) x))
                     (return-from division nil)
                     (/!! y x)))))))))
```

Here **return-from** is used to exit from the **division** block if the value of **x** in any processor is zero. When the **with-css-saved** macro is entered, it saves the state of the currently selected set. When the code enclosed within the **with-css-saved** exits for any reason, either normally or via a call to an non-local exit operator like **return-from**, the currently selected set is restored to its original state.

See the dictionary entry for **with-css-saved** for more information.

Implementation Note:

If the last *body* form is either a ***all** or a ***when** form, then the inner form does not save/ restore the state of the current selected set. This is mainly an optimization feature—it does not change the semantics of your code.

REFERENCES

See also the related operators

***case**	**case!!**	***cond**	**cond!!**	***ecase**	**ecase!!**
***if**	**if!!**	***unless**	***when**	**with-css-saved**	

allocate!! *[Macro]*

Allocates a global pvar.

SYNTAX

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

ARGUMENTS

pvar-initial-value	Pvar expression. If supplied, is value with which global pvar is initialized. If not supplied, a pvar with undefined values is created.
name	Symbol. If supplied, stored as the symbolic name of the allocated pvar.
type	Data type specification. If supplied, determines the data type of the allocated pvar. Must be compatible with data type of *pvar-initial-value* argument. If not supplied, a general mutable pvar is created.

RETURNED VALUE

global-pvar	The created global pvar is returned.

SIDE EFFECTS

The returned pvar is allocated on the heap.

DESCRIPTION

This operation creates a global pvar with the specified *pvar-initial-value*, *name*, and *type*. Global pvars are deallocated during a call to ***cold-boot**, and are *not* automatically reallocated, as are permanent pvars created by ***defvar**.

EXAMPLES

Global pvars of any data type may be allocated on the heap using **allocate!!**:

```
(setq a (allocate!! (!! 5)))

(setq b (allocate!! (evenp!! (random!! (!! 2)))
                    'new-pvar 'boolean-pvar))

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

(ppp heap-pvar :end 2)
=> #(1 2 3) #(1 2 3)
```

The following example shows how **allocate!!** may be used to allocate pvars within any VP set, and also how **allocate!!** is useful for creating an unspecifed number of global pvars on demand.

```
(def-vp-set fred (list *minimum-size-for-vp-set*))

(defvar list-of-pvars nil)

(defun main
   (*with-vp-set fred
      (loop
         (process-data)
         (when (extra-pvar-needed)
            (push (allocate!!   (!! 0) nil
                                '(pvar (unsigned-byte 32)))
             list-of-pvars)))))
```

By defining the **list-of-pvars** with **allocate!!**, the global pvars pushed onto the list may be explicitly deallocated with the ***deallocate** operator whenever they are no longer needed.

NOTES

Usage Note:

The **allocate!!** macro is intended to be called within user code, not at top level. It acts much like the **malloc** operator in the C language, in allowing the programmer to dynamically allocate CM memory within a program. Pvars allocated using **allocate!!** are automatically deallocated during a ***cold-boot**. It is an error to attempt to reference a global pvar deallocated by ***cold-boot**.

Language Note:

Global pvars and permanent pvars are allocated on the CM heap. In contrast to global pvars, which are allocated by **allocate!!** and deallocated with ***deallocate**, permanent pvars are allocated by ***defvar** and must be deallocated by the function ***deallocate–*defvars.**

A global pvar created with **allocate!!** is simply returned. A permanent pvar created with ***defvar** is bound to a global variable. Permanent pvars are reallocated during a call to ***cold–boot**; global pvars are simply deallocated.

REFERENCES

See also the pvar allocation and deallocation operations

array!!		
***deallocate**	***deallocate–*defvars**	***defvar**
front–end!!	***let**	***let***
make–array!!	**typed–vector!!**	**vector!!**
!!		

See the *Lisp glossary for definitions of the different kinds of pvars that are allocated on the CM stack and heap.

allocate–processors–for–vp–set [*Function*]
allocate–vp–set–processors [*Function*]

Instantiates the specified flexible VP set, allocating virtual processors according to the supplied dimensions or geometry.

SYNTAX

allocate–processors–for–vp–set *vp–set dimensions* **&key :geometry**

ARGUMENTS

vp–set	Flexible VP set. Virtual processor set defined with **def–vp–set.**
dimensions	Integer list or **nil**. Size of dimensions with which to instantiate *vp–set*. Must be **nil** if *geometry* argument is supplied.
:geometry	Geometry object obtained by calling the function **create–geometry**. Defines geometry of *vp–set*.

RETURNED VALUE

nil	Evaluated for side effect.

SIDE EFFECTS

Defines geometry of and instantiates *vp–set*, and allocates any associated pvars.

DESCRIPTION

This function is used during program execution to instantiate a flexible VP set. A flexible VP set is a VP set that has been defined by calling **def–vp–set** without supplying specific dimensions or 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 example, VP set geometries might depend on characteristics of data that are read from a file during program execution.

It is an error to invoke **allocate–processors–for–vp–set** before ***cold–boot** has been invoked, or to pass a fixed-size VP set as an argument.

The argument *vp–set* must be a flexible VP set defined by a call to the **def–vp–set** macro in which the *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 a **:geometry** keyword argument is supplied, 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** for more details.)

EXAMPLES

This example shows how **allocate–processors–for–vp–set**, along with its companion function **deallocate–processors–for–vp–set**, may be used to instantiate a flexible VP set several times with a different geometry at each invocation.

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

(defun process-files (&rest diskfiles)
  (*cold-boot)
  ;;; at this point, disk-data-pvar has no memory allocated
  ;;; on the CM
  (dolist (file diskfiles)
    (let ((elements (read-number-of-elements-in file)))
      (allocate-processors-for-vp-set disk-data
         (list (next-power-of-two->= elements)))
      ;;; now disk-data-pvar has CM memory allocated
      (let ((array-of-data (read-data-from-disk file)))
        (array-to-pvar array-of-data disk-data-pvar
                       :cube-address-end elements)
        (process-data-in-cm disk-data disk-data-pvar))
      (deallocate-processors-for-vp-set disk-data))))
```

NOTES

The function **allocate–vp–set–processors** is an obsolete alias for **allocate–processors–for–vp–set**, and behaves identically.

REFERENCES

See also the following VP set definition and deallocation operators:

def–vp–set **create–vp–set** **let–vp–set**
deallocate–def–vp–sets **deallocate–vp–set**

See also the following geometry definition operator:

create–geometry

The following math utilities are useful in defining the size of VP sets:

next–power–of–two–>= **power–of–two–p**

See also the following flexible VP set operators:

deallocate–vp–set–processors **deallocate–processors–for–vp–set**
set–vp–set–geometry **with–processors–allocated–for–vp–set**

allocated–pvar–p *[Function]*

Tests whether a pvar has CM memory allocated for it and, if so, whether it is on the stack or the heap.

SYNTAX

allocated–pvar–p *pvar*

ARGUMENTS

pvar	Pvar expression.

RETURNED VALUE

allocated–p	A symbol. If *pvar* is allocated, either **:stack** or **:heap** is returned, indicating where it is allocated. If *pvar* is not allocated then **nil** is returned.

SIDE EFFECTS

None.

DESCRIPTION

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
```

REFERENCES

See also the following general pvar information operators:

describe–pvar	**pvar–exponent–length**	
pvar–length	**pvar–location**	**pvar–mantissa–length**
pvar–name	**pvarp**	**pvar–plist**
pvar–type	**pvar–vp–set**	

alpha–char–p!! [*Function*]

Performs a parallel test for alphabetic characters on the supplied pvar.

SYNTAX

alpha–char–p!! *character–pvar*

ARGUMENTS

character–pvar Character pvar. Tested in parallel for alphabetic characters.

RETURNED VALUE

alpha–charp–pvar Temporary boolean pvar. Contains the value **t** in each active processor where the corresponding value of *character–pvar* is an alphabetic character. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The function **alpha–char–p!!** is a parallel character predicate. It returns a temporary pvar containing **t** in each active processor where the corresponding value of *character–pvar* is an alphabetic character, and **nil** in all other active processors. The function **alpha–char–p!!** provides the same functionality for character pvars that the Common Lisp character predicate **alpha–char–p** provides for scalar characters.

EXAMPLES

Alphabetic characters are all of the characters between **#\A** and **#\Z**, **#\a** and **#\z** inclusive. The pvar that **alpha–char–p!!** returns contains **t** in each processor where the corresponding value of *character–pvar* is one of these characters.

For example, if **char–pvar** contains the values **#\A, #\newline, #\Q, #\z, #\5, #\!**, etc., then the pvar returned by

```
(alpha-char-p!! char-pvar)
```

will contain the values **t**, **nil**, **t**, **t**, **nil**, **nil**, etc.

The function **alpha–char–p!!** is most useful in combination with the processor selection operators. For example, if **text–pvar** is a character pvar representing a string of text, then

```
(*when (alpha-char-p!! text-pvar)
   (*sum (!! 1)))
```

returns the number of alphabetic characters in the string. Here, the macro ***when** is used to select only those processors containing an alphabetic character. Then, ***sum** is applied to the constant pvar **(!! 1)** to return a count of the number of selected processors.

alphanumericp!! [*Function*]

Performs a parallel test for alphanumeric characters on the supplied pvar.

SYNTAX

alphanumericp!! *character–pvar*

ARGUMENTS

character–pvar Character pvar. Tested in parallel for alphanumeric characters.

RETURNED VALUE

alphanumericp–pvar

Temporary boolean pvar. Contains the value **t** in each active processor where the corresponding value of *character–pvar* is an alphanumeric character. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The function **alphanumericp!!** is a parallel character predicate. It returns a temporary pvar containing **t** in each active processor where the corresponding value of *character–pvar* is an alphabetic or numeric character, and **nil** in all other active processors. Thus, the following forms are equivalent:

```
(alphanumericp!! character-pvar)
<=>
(or!! (alpha-char-p!! character-pvar)
      (digit-char-p!! character-pvar))
```

The function **alphanumericp!!** provides the same functionality for character pvars that the Common Lisp character predicate **alphanumericp** provides for scalar characters.

EXAMPLES

Alphanumeric characters are all of the characters between **#\A** and **#\Z**, **#\a** and **#\z**, and **#\0** and **#\9** inclusive. The pvar that **alphanumericp!!** returns contains **t** in each processor where the corresponding value of *character-pvar* is one of these characters. For example, if **char-pvar** contains the values **#\A**, **#\newline**, **#\Q**, **#\z**, **#\5**, **#\!**, etc., then the pvar returned by

```
(alphanumericp!! char-pvar)
```

will contain the values **t**, **nil**, **t**, **t**, **t**, **nil**, etc.

The function **alphanumericp!!** is most useful in combination with the processor selection operators. For example, if **text-pvar** is a character pvar representing a string of text, then

```
(*when (alphanumericp!! text-pvar)
   (*sum (!! 1)))
```

returns the number of alphanumeric characters in the string. The macro ***when** is used to select only those processors containing an alphanumeric character, and then ***sum** is applied to the constant pvar **(!! 1)** to return a count of the number of selected processors.

amap!! *[Function]*

Maps a function in parallel over a set of array pvars.

SYNTAX

amap!! *operator array–pvar* **&rest** *array–pvars*

ARGUMENTS

operator — Parallel function. Must accept the same number of arguments as the number of *array–pvar* arguments supplied.

array–pvar, array–pvars — Array pvars. Combined in parallel using *operator*.

RETURNED VALUE

result–pvar — Temporary array pvar. In each active processor, contains an array whose value in each element is the result of combining the corresponding elements of the arrays in the *array–pvars* using the specified *operator*.

SIDE EFFECTS

The resulting pvar is allocated on the stack.

DESCRIPTION

The **amap!!** function maps the supplied *operator* over the supplied array pvars. The *operator* is applied in turn to each set of elements having the same row-major index in the supplied *array–pvars*. Thus, the *n*th time *function* is called, it is applied to a list containing the *n*th element in row-major order from each of the *array–pvars*.

The returned array pvar contains in each active processor an array whose value in any given element is the result of applying *operator* to the values of the corresponding elements of the arrays in the supplied *array–pvars*.

The *Lisp function **amap!!** is similar to the Common Lisp function **map**, but while **map** works only on vectors, **amap!!** works on any type of array pvar. The **amap!!** function requires no result type specification, as **map** does, because the result is always returned as an array pvar.

For vectors, the **amap!!** function behaves much like the **map** function in accepting vector pvar arguments of different element sizes and in limiting the mapping operation to the length of the shortest vector pvar supplied. For all other types of array pvars, however, **amap!!** expects the array sizes of the supplied *array-pvars* to be identical.

EXAMPLES

The **amap!!** can be used to emulate vector operators such as the parallel vector addition function **v+!!**. For example, **v+!!** is equivalent to calling **amap!!** with an operator of **'+!!**. Thus:

```
(v+!! a b) <==> (amap!! '+!! a b)
```

As another example, if **y** and **x** are vector pvars of length *n*, then

```
(*set y (amap!! 'log!! (amap!! 'cos!! x)))
```

is equivalent to

```
(dotimes (j n)
   (*setf (aref!! y (!! j))
          (log!! (cos!! (aref!! x (!! j))))))
```

REFERENCES

Also see the function ***map**, which behaves somewhat like **amap!!** but does not return a value.

*and [*Defun]

Takes the logical AND of all active values in a pvar, returning a scalar value.

SYNTAX

***and** *pvar–expression*

ARGUMENTS

pvar–expression Pvar expression. Pvar to which global AND is applied.

RETURNED VALUE

and–scalar Scalar boolean value. The logical AND of the values in *pvar*.

SIDE EFFECTS

None.

DESCRIPTION

The ***and** function is a global operator. It returns a scalar value of **t** if the value of *pvar–expression* in every active processor is non-**nil**, and returns **nil** otherwise.

If there are no active processors, this function returns **t**.

EXAMPLES

The function ***and** can be used to determine whether any value of a pvar fails a given predicate. For example,

```
(*and (evenp!! numeric-pvar))
```

returns **t** if every value of **numeric–pvar** is even, and **nil** if any value is odd.

The following is a simple function definition using ***and**:

```
(*defun *t (pvar) (*and (eql!! pvar t!!)))
```

The function ***t** returns **t** if and only if its **pvar** argument is equal to **t!!**, that is, if it contains the value **t** in every processor.

The function ***and** is also useful for determining whether an operation has been performed on all values of a pvar. For example, the function defined by

```
(defun value-list (pvar)
   (*let ((checked-pvar nil!!))
      (do   ((return-list nil))
            ((*and checked-pvar) return-list)
        (*when (not!! checked-pvar)
          (let ((minumum (*min pvar)))
             (push minimum return-list)
             (*when (=!! pvar (!! minimum))
                (*set checked-pvar t!!)))))))
```

returns a list of the numeric values contained in **pvar** in all of the currently active processors. The variable **checked–pvar**, initially set to **nil!!**, indicates which of the currently selected processors have already been checked.

Each time around the **do** loop, ***when** is used to select all active processors which have not been checked. The minimum value contained in these processors is found using ***min**, and pushed onto **return–list**. The variable **checked–pvar** is modified, using ***set**, to indicate that all processors having this value have been checked.

Each time around the loop, **checked–pvar** is checked using ***and**. When **(*and checked–pvar)** returns **t**, indicating that all of the currently active processors have been checked, the loop exits, and **return–list**, the list of collected values, is returned.

REFERENCES

See also the related global operators:

***integer–length**	***logand**	
***logior**	***logxor**	***max**
***min**	***or**	***sum**
***xor**		

See also the related logical operators:

and!! **not!!** **or!!** **xor!!**

and!! [*Macro*]

Performs a parallel logical AND operation in all active processors.

SYNTAX

and!! &rest *pvar–exprs*

ARGUMENTS

pvar–exprs	Pvar expressions. Pvars to which parallel AND is applied.

RETURNED VALUE

and–pvar	Temporary pvar. In each active processor, contains the value **nil** if any of the *pvar–exprs* evaluate to **nil** in that processor; contains the value of the last of the *pvar–exprs* otherwise.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The **and!!** function performs a parallel logical AND operation. In all active processors, it evaluates each of the supplied *pvar–exprs* in order from left to right. As soon as one of the *pvar–exprs* evaluates to **nil** in a processor, that processor is removed from the currently selected set for the remainder of the **and!!**.

The temporary pvar returned by **and!!** contains the value of the last of the *pvar–exprs* in those processors for which each of the previous *pvar–exprs* evaluated to a non-**nil** value, and **nil** in all other active processors. If no *pvar–exprs* are supplied, the pvar **t!!** is returned.

The function **and!!** provides functionality for boolean pvars similar to that which the Common Lisp function **and** provides for boolean values.

EXAMPLES

The **and!!** function can be used either as a straightforward logical operator or as a means of controlling evaluation. For example, the pvar returned by

```
(and!! (integerp!! numeric-pvar)
       (>=!! numeric-pvar (!! -5))
       (<=!! numeric-pvar (!! 5)))
```

contains **t** in each active processor for which the value of **numeric-pvar** is an integer between –5 and 5, inclusive, and **nil** in all other active processors. We could add **numeric-pvar** as the final argument, so:

```
(and!! (integerp!! numeric-pvar)
       (>=!! numeric-pvar (!! -5))
       (<=!! numeric-pvar (!! 5))
       numeric-pvar)
```

This now returns a pvar containing the original value from **numeric-pvar** in each processor where that value is an integer between –5 and 5, and **nil** in all other active processors.

Because **and!!** controls the selected set in which its arguments are evaluated, it can be used to control evaluation of pvar expressions. The expression

```
(if!!  (and!! (integerp!! data-pvar)
              (plusp!! data-pvar))
       (sqrt!! data-pvar))
```

returns a pvar whose value in each active processor is the square-root of the corresponding value of **data-pvar**, if that value is a positive integer, and **nil** otherwise.

NOTES

Language Note:

Remember that **and!!** changes the currently selected set as it evaluates its arguments. This can have unwanted side effects in code that depends on unchanging selected sets, particularly code involving communication operators, such as **scan!!**.

For example, the expressions

```
(ppp (and!! (evenp!! (self-address!!))
            (<!! (scan!! (self-address!!) '+!!) (!! 3)))
     :end 8)
T NIL T NIL NIL NIL NIL NIL

(ppp (and!! (<!! (scan!! (self-address!!) '+!!) (!! 3))
            (evenp!! (self-address!!)))
     :end 8)
T NIL NIL NIL NIL NIL NIL NIL
```

exemplify a case in which using **and!!** may cause a non-intuitive result because of its deselection properties. In the first expression, the **scan!!** operation is performed only in the even processors. In the second expression, the **scan!!** operation is performed in all processors, resulting in a different set of displayed values.

This is the result of **and!!** deselecting those processors that fail any clause before executing the next clause. One can avoid this in the following manner:

```
(*let ((b1 (evenp!! (self-address!!)))
       (b2 (<!! (scan!! (self-address!!) '+!!) (!! 3))))
  (declare (type boolean-pvar b1 b2))
  (and!! b1 b2))
```

REFERENCES

See also the related global operators:

***and**	***integer–length**	***logand**
***logior**	***logxor**	***max**
***min**	***or**	***sum**
***xor**		

See also the related logical operators:

not!!	**or!!**	**xor!!**

*apply [Macro]

Applies a parallel function defined with ***defun** to a set of arguments.

SYNTAX

***apply** *function* **&rest** *args*

ARGUMENTS

function	*Lisp function.
args	Set of scalar or pvar values. Arguments to which *function* is applied. Last argument supplied must be a list.

RETURNED VALUE

result	Scalar or pvar. Result of applying *function* to the supplied *args*.

SIDE EFFECTS

None aside from those produced by *function*.

DESCRIPTION

This is the parallel equivalent of the Common Lisp **apply** operator, but is intended to be used with functions defined using ***defun**. Each of the supplied *args* except the last are collected into a list, which is then appended to the last of the *args*. The *function* is applied to the resulting list.

The ***apply** operator can be used to call functions defined with **defun**, as well, but it is more efficient to use **apply** instead.

EXAMPLES

```
(*defun percent-difference!! (pvarl pvar2)
    (*!! (/!! (-!! pvar2 pvarl) pvarl) (!! 100)))

(*apply 'percent-difference!!
                 (!! 2) (list (!! 4))) <=>  (!! 100.0)
(*apply 'percent-difference!!
                 (list (!! 5) (!! 2))) <=>  (!! -60.0)
```

NOTES

It is an error to use the Common Lisp **apply** operator with a function defined using ***defun.** Also, just as **apply** cannot be applied to macros, so ***apply** cannot be applied to macros with the exception of operations defined by ***defun**. (Observant readers will notice that an operation defined by ***defun** actually *is* a macro in disguise—see the dictionary entry for ***defun** for more information.)

It is legal to provide a **lambda** form as the *function* argument to ***apply**. However, in this case there is no difference between using **apply** or using ***apply**, and using **apply** is preferred for clarity.

REFERENCES

See also the following related operations:

***defun**	***funcall**	
***trace**	**un*defun**	***untrace**

aref!! [*Function*]

Performs a parallel array reference on the supplied array pvar.

SYNTAX

aref!! *array–pvar* **&rest** *subscript–pvars*

ARGUMENTS

array–pvar	Array pvar. Pvar from which values are referenced.
subscript–pvars	Integer pvars. Non-negative indices of the array location to be referenced in each processor. The number of *subscript–pvars* must equal the rank of *array–pvar*.

RETURNED VALUE

value–pvar	Temporary pvar. Value retrieved in each processor.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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*.

One *subscript–pvar* argument must be given for each dimension of *array–pvar*. Each *subscript–pvar* must contain non-negative integers within the legal range of coordinates for that dimension.

EXAMPLES

A sample call to **aref!!** is

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

which returns a pvar containing in each processor a copy of the element (1,4) of **2by5-array-pvar** that is stored in that processor. An actual example of an array reference is

```
(*defvar array-pvar (!! #2A((1 2 3) (4 5 6))))

(aref!! array-pvar (!! 0) (!! 2))   <=>   (!! 3)
```

Here, the element (0,2) of the **array-pvar** in each processor is 3, so the call to **aref!!** with constant *subscript-pvar* arguments (pvars having the same value in each processor) returns a pvar containing the value 3 in each processor.

The ***setf** operator may be used with **aref!!** to modify array locations in parallel. For example,

```
(*setf (aref!! array-pvar (!! 0) (!! 2))  (!! 9))
```

The *subscript-pvar* arguments to **aref!!** can contain different values in each processor. This is known as non-constant array indexing. An example of non-constant indexing is

```
(*proclaim '(type (vector-pvar single-float 2) xyzzy))
(*defvar xyzzy)

(defun non-constant-indexing-example ()
  (*setf (aref!! xyzzy (!! 0)) (!! 1.0))
  (*setf (aref!! xyzzy (!! 1)) (!! -1.0))
  (ppp (aref!! xyzzy
             (if!! (evenp!! (self-address!!)) (!! 0) (!! 1)))
       :end 8))

(non-constant-indexing-example)
1.0 -1.0 1.0 -1.0 1.0 -1.0 1.0 -1.0
```

NOTES

Performance Note:

In general, especially for large arrays, the CM-2 implementation of non-constant indexing can be very slow. See ***sideways-array** and **sideways-aref!!** for a means of using the CM-2 architecture to do fast non-constant indexing into arrays.

Usage Note:

In most cases, it is unnecessary to use the operator **alias!!** in combination with **aref!!** to prevent the copying of pvars, because the *Lisp compiler is able to recognize and optimize cases where this copying is unnecessary.

For example,

```
(*proclaim '(type  (array-pvar single-float (4))
                   data-array-pvar))
(*defvar data-array-pvar (!! #(0.0 1.0 2.0 3.0)))

(defun bad-example (x)
   (*set (the single-float-pvar x)
      (+!!   (alias!! (aref!! data-array-pvar (!! 0)))
            (alias!! (aref!! data-array-pvar (!! 1))))))
```

It is unnecessary to use **alias!!** to avoid having a copy of the data in elements 0 and 1 of **data-array-pvar** being made. As long as the *Lisp compiler is compiling the code, then

```
(defun good-example(x)
   (*set (the single-float-pvar x)
      (+!!   (aref!! data-array-pvar (!! 0))
            (aref!! data-array-pvar (!! 1)))))
```

is equivalent and will not result in any temporary pvars being used. In general there is no need to use **alias!!** when performing array accessing except in certain special cases that are discussed under the dictionary entry for **alias!!**.

REFERENCES

See also the related array-referencing operations:

row-major-aref!!
row-major-sideways-aref!! **sideways-aref!!**

The following operations convert arrays to and from sideways orientation:

***processorwise** ***sideways-array** ***slicewise**

array!!   [*Function*]

Creates and returns an array pvar. In each active processor, an array of the specified dimensions is created and initialized with corresponding values from the specified pvars.

SYNTAX

array!! *dimensions* **&rest** *content–pvars*

ARGUMENTS

dimensions	Integer list. Specifies dimensions of array to store in each processor.
content–pvars	Pvars. In each processor, specify, in row-major order, the values to be stored in that processor's array. The number of *content–pvars* supplied must match the number of array elements specified by *dimensions*.

RETURNED VALUE

array–pvar	Temporary array pvar. Contains in each active processor an array of the specified *dimensions* containing the values of the *content–pvars*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The **array!!** function creates an array pvar with the specified *dimensions*, initialized to contain the values of the specified *content–pvars*.

The returned *array–pvar* consists of an array in each active processor. The values of each processor's array elements are copied, in row-major order, from the corresponding values of each supplied *content–pvar*.

EXAMPLES

```
(array!! '(2 2)  (!! 0) (!! 1)
                 (!! 2) (!! 3))
<=>
(!! #2A((0 1) (2 3)))
```

NOTES

The standard rules of coercion are used to determine the element type of the new parallel array. Thus, a mixture of integer and floating-point elements yields a floating-point result. A mixture of floating-point and complex elements yields a complex result. An error is signaled if the data types present are not all compatible. For instance, a string-char element and a floating-point element are not compatible.

REFERENCES

See also the pvar allocation and deallocation operations

allocate!!

***deallocate**	***deallocate–*defvars**	***defvar**
front–end!!	***let**	***let***
make–array!!	**typed–vector!!**	**vector!!**

!!

*array–dimension [*Defun]
array–dimension!! [Function]

Return the length of an array pvar along a specified dimension.

SYNTAX

***array–dimension** *array–pvar dimension*
array–dimension!! *array–pvar dimension–pvar*

ARGUMENTS

array–pvar	Array pvar.
dimension	Scalar integer. Index of an array dimension of *array–pvar*.
dimension–pvar	Integer pvar. In each processor, the index of an array dimension of *array–pvar*.

RETURNED VALUE

For ***array–dimension**:

length	Scalar integer. Length of *array–pvar* along specified *dimension.*

For **array–dimension!!**:

length–pvar	Temporary integer pvar. Contains in each active processor the length of *array–pvar* along the specified *dimension.*

SIDE EFFECTS

For **array–dimension!!**, the returned pvar is allocated on the stack.

DESCRIPTION

The ***array–dimension** function returns an unsigned integer equal to the size of the dimension of *array–pvar* referenced by *dimension.* The argument *dimension* must be an unsigned integer between 0 and 1 less than the rank of *array–pvar.*

The **array–dimension!!** function returns a pvar containing, in each processor, an unsigned integer equal to the length of the *dimension–pvar* dimension of *array–pvar.* The argument *dimension–pvar* must be a pvar containing, in each processor, an unsigned integer less than the rank of *array–pvar.*

EXAMPLES

```
(*defvar my-array-pvar (array!! '(2 1) (!! 0) (!! 1)))

(*array-dimension my-array-pvar 0) => 2
(*array-dimension my-array-pvar 1) => 1

(*defvar array-pvar (array!! '(2 1) (!! 0) (!! 1)))

(ppp (array-dimension!! array-pvar
                        (mod!! (self-address!!) (!! 2)))
     :end 12)
2 1 2 1 2 1 2 1 2 1 2 1 2 1
```

REFERENCES

See also the related array pvar information operators:

***array–dimensions**	**array–dimensions!!**
***array–element–type**	**array–in–bounds–p!!**
***array–rank**	**array–rank!!**
***array–total–size**	**array–total–size!!**
array–row–major–index!!	**sideways–array–p**

*array–dimensions [*Defun]
array–dimensions!! [Function]

Return a list of the lengths of each dimension of an array pvar.

SYNTAX

***array–dimensions** *array–pvar*
array–dimensions!! *array–pvar*

ARGUMENTS

array–pvar Array pvar.

RETURNED VALUE

For ***array–dimensions**:

lengths–list Scalar integer list. Lengths of the dimensions of *array–pvar*.

For **array–dimensions!!**:

lengths–pvar Temporary vector pvar. In each active processor, contains a vector enumerating the lengths of the dimensions of *array–pvar*.

SIDE EFFECTS

For **array–dimensions!!**, the returned pvar is allocated on the stack.

DESCRIPTION

The ***array–dimensions** function returns a front-end list enumerating the dimensions of *array–pvar*. This list is of length **(*array–rank array–pvar)**.

The **array–dimensions!!** function returns a vector pvar containing, in each processor, a vector whose *n*th element is the length of the *n*th dimension of *array–pvar*.

EXAMPLES

```
(*set my-array-pvar (array!! '(2 1) (!! 0) (!! 1)))

(*array-dimensions my-array-pvar)      =>    (2 1)
(array-dimensions!! my-array-pvar)     <=>   (!! #(2 1))
```

NOTES

By definition, all arrays in an array pvar have the same size and shape. Thus, the pvar returned by **array–dimensions!!** will always have the same value in all processors.

REFERENCES

See also the related array pvar information operators:

***array–dimension**	**array–dimension!!**
***array–element–type**	**array–in–bounds–p!!**
***array–rank**	**array–rank!!**
***array–total–size**	**array–total–size!!**
array–row–major–index!!	**sideways–array–p**

*array–element–type [*Defun]

Returns type specifier for the elements of an array pvar.

SYNTAX

***array–element–type** *array–pvar*

ARGUMENTS

array–pvar	Array pvar. Pvar for which element type is to be returned.

RETURNED VALUE

type–spec	Type specifier for elements of *array–pvar.*

DESCRIPTION

This function returns a front-end type specifier for the elements of *array–pvar.*

EXAMPLES

```
(*array-element-type (array!! '(1 1) (!! 0)))
=>  (PVAR (UNSIGNED-BYTE 1))
```

REFERENCES

See also the related array pvar information operators:

***array–dimension**	**array–dimension!!**
***array–dimensions**	**array–dimensions!!**
array–in–bounds–p!!	
***array–rank**	**array–rank!!**
***array–total–size**	**array–total–size!!**
array–row–major–index!!	**sideways–array–p**

array–in–bounds–p!! [*Function*]

Tests in parallel whether array subscripts are within the bounds of an array pvar.

SYNTAX

array–in–bounds–p!! *array–pvar* **&rest** *subscript–pvars*

ARGUMENTS

array–pvar	Array pvar.
subscript–pvars	Integer pvars. Subscripts to be checked against bounds of *array–pvar*.

RETURNED VALUE

in–boundsp–pvar	Temporary boolean pvar. Contains **t** in every processor where the *subscript–pvars* represent a valid reference to *array–pvar*. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function returns a boolean pvar with **t** in every processor where the values of the supplied *subscript–pvars* represent a valid reference to *array–pvar* and **nil** elsewhere.

EXAMPLES

```
(*set my-array-pvar (array!! '(1 1) (!! 0)))

(array-in-bounds-p!! my-array-pvar (!! 0) (!! 0))  <=>  t!!
(array-in-bounds-p!! my-array-pvar (!! 2) (!! 0))  <=>  nil!!
```

REFERENCES

See also the related array pvar information operators:

***array-dimension**	**array-dimension!!**
***array-dimensions**	**array-dimensions!!**
***array-element-type**	
***array-rank**	**array-rank!!**
***array-total-size**	**array-total-size!!**
array-row-major-index!!	**sideways-array-p**

See also the related array-referencing operations:

aref!!	**row-major-aref!!**
row-major-sideways-aref!!	**sideways-aref!!**

*array–rank [*Defun]
array–rank!! [Function]

Return the number of dimensions of an array pvar.

SYNTAX

***array–rank** *array–pvar*
array–rank!! *array–pvar*

ARGUMENTS

array–pvar Array pvar.

RETURNED VALUE

For ***array–rank**:

rank Integer. Number of dimensions of *array–pvar*.

For **array–rank!!**:

rank–pvar Temporary integer pvar. Contains in each active processor the rank, or number of dimensions, of *array–pvar*.

SIDE EFFECTS

For **array–rank!!**, the returned pvar is allocated on the stack.

DESCRIPTION

The ***array–rank** function returns an unsigned integer equal to the number of dimensions in *array–pvar*.

The **array–rank!!** function returns a pvar containing, in each processor, an unsigned integer equal to the number of dimensions in *array–pvar.*

EXAMPLES

```
(*array-rank (array!! '(2 1) (!! 0) (!! 1)))  =>  2

(array-rank!! pvar)  <=>  (!! (*array-rank pvar))
```

NOTES

By definition, all arrays in an array pvar have the same size and shape. Thus, the pvar returned by **array–rank!!** has the same value in all processors.

REFERENCES

See also the related array pvar information operators:

***array–dimension**	**array–dimension!!**
***array–dimensions**	**array–dimensions!!**
***array–element–type**	**array–in–bounds–p!!**
***array–total–size**	**array–total–size!!**
array–row–major–index!!	**sideways–array–p**

array–row–major–index!! [*Function*]

Converts array subscripts to row major indices in parallel.

SYNTAX

array–row–major–index!! *array–pvar* **&rest** *subscript–pvars*

ARGUMENTS

array–pvar	Array pvar.
subscript–pvars	Integer pvars. Must contain subscripts valid for *array–pvar.*

RETURNED VALUE

indices–pvar	Temporary integer pvar. In each processor, contains the corresponding row major index in *array–pvar* for the set of subscripts in the *subscript–pvars.*

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

In each processor, this function converts the array pvar subscripts contained in *subscript–pvars* into row-major indices for *array–pvar.*

The *subscript–pvars* must contain 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 *subscript–pvars* arguments must equal the rank of *array–pvar.*

In each processor the returned *indices–pvar* contains a single integer, the row-major index of the array element specified by the values of the *subscript–pvars.* This pvar of row-major indicies may be used to access the *array–pvar* via the function **row–major–aref!!.**

EXAMPLES

Consider a two-dimensional array pvar, as defined by

```
(*defvar arr!! (!! #2A((10 30) (20 40))))
```

The row-major index of each element in **arr!!** can be determined as follows:

```
(ppp a :end 4)   =>  0 1 0 1
(ppp b :end 4)   =>  0 0 1 1
(ppp (array-row-major-index!! arr!! a b)
   :end 4)       =>  0 2 1 3
```

That the row-major indices are independant of the contents of the array elements can be see by evaluating the expression

```
(ppp (aref!! arr!! a b) :end 4)  =>  10 20 30 40
```

REFERENCES

See also the related array pvar information operators:

***array-dimension**	**array-dimension!!**
***array-dimensions**	**array-dimensions!!**
***array-element-type**	**array-in-bounds-p!!**
***array-rank**	**array-rank!!**
***array-total-size**	**array-total-size!!**
sideways-array-p	

array–to–pvar [*Defun]

In send (cube) address order, copies values from a front-end vector to a pvar.

SYNTAX

array–to–pvar *source–array* **&optional** *dest–pvar*
&key **:array–offset**
:start **:cube–address–start**
:end **:cube–address–end**

ARGUMENTS

source–array	Front-end vector. Array from which values are copied.
dest–pvar	Pvar. An allocated pvar in any VP set, into which values are stored. If not supplied, **array–to–pvar** creates a temporary pvar in the current VP set.
:array–offset	Integer. Offset into *source–array* of first value to copy. Default is 0.
:start	Send address. Processor at which copying starts. Default is 0.
:end	Send address. Processor at which copying ends. Default is ***number–of–processors–limit***.
:cube–address–start, :cube–address–end	Obsolete aliases for **:start** and **:end** keywords, retained for software compatibility only.

RETURNED VALUE

dest–pvar	The destination pvar, containing values copied from *source–array*. If a *dest–pvar* argument is supplied, values are copied into it. If not, a temporary pvar is created and returned.

SIDE EFFECTS

The contents of *source–array*, beginning at the element specified by *array–offset*, are copied into *dest–pvar*. All values of *dest–pvar* from **:start** to **:end** are modified, regardless of the currently selected set. If the *dest–pvar* argument is not supplied, a temporary pvar is allocated on the stack.

DESCRIPTION

This function copies data from *source–array* to *dest–pvar* in send-address order. The *source–array* must be one-dimensional. If a *dest–pvar* is not provided, **array–to–pvar** creates a temporary destination pvar. If a temporary destination pvar is created, its value in processors to which **array–to–pvar** did not write is undefined.

It is legal for *source–array* to contain more elements than can be stored in *dest–pvar*. The extra elements are ignored. It is an error, however, for *source–array* to contain fewer elements than are needed to fill *dest–pvar*.

This function is especially useful for copying data into the CM. It is much faster than setting pvar elements individually using ***setf** and **pref**.

EXAMPLES

After the following forms are evaluated,

```
(*defvar pvar)
(setq array (make-array      *number-of-processors-limit*
                             :initial-element 3))
(array-to-pvar array pvar)
```

The value of **pvar** is **(!! 3)**.

NOTES

Usage Note:

It is an error to supply both a **:cube–address–start** and a **:start** argument. Likewise, it is an error to supply both a **:cube–address–end** and a **:end** argument.

Performance Note:

This operation is fastest when pvars of a specific non-aggregate type are used, slower when general pvars are used, and very slow if aggregate pvars are used. The examples below shows how to move aggregate data efficiently into the CM.

The following expressions define a ***defstruct** type and create a structure pvar of that type.

```
(*defstruct foo
  (a 0 :type t :cm-type (pvar (unsigned-byte 32)))
  (b 0.0 :type t :cm-type (pvar single-float))
  )

(*proclaim '(type (pvar foo) a-foo-pvar))
(*defvar a-foo-pvar)
```

n the first example, an array of structure objects of type **foo** is created on the front end, and then copied in one operation to a structure pvar on the CM. This method of transferring data is very slow, but is relatively straightforward.

```
(defvar a-foo-array
  (make-array *number-of-processors-limit*
    :element-type 'foo))

(defun init-a-foo-array ()
  (dotimes (j *number-of-processors-limit*)
    (setf (aref a-foo-array j) (make-foo))
    ))
(defun move-a-foo-array-data-from-front-end-to-cm ()
  (array-to-pvar a-foo-array a-foo-pvar)
  )
```

The next example is very fast, although it is somewhat non-intuitive. The expressions below create a single front-end structure object, and initialize its slots with arrays of values that will form the slot values of the structure pvar on the CM. Moving the data to the CM involves a separate array transfer for each slot, copying the array of elements for that slot to the structure pvar on the CM.

```
;;; create single front-end structure object
(defvar a-foo (make-foo))
```

```
;;; initialize the object's slots with arrays
;;; instead of single values
(defun init-a-foo ()
  (setf (foo-a a-foo)
    (make-array *number-of-processors-limit*
                :element-type '(unsigned-byte 32)))
  (setf (foo-b a-foo)
    (make-array *number-of-processors-limit*
                :element-type 'single-float)))

;;; perform one array-to-pvar transfer for each slot
;;; (note use of alias!! to prevent slot copying)
(defun move-a-foo-data-from-front-end-to-cm ()
  (array-to-pvar (foo-a a-foo)
    (alias!! (foo-a!! a-foo-pvar)))
  (array-to-pvar (foo-b a-foo)
    (alias!! (foo-b!! a-foo-pvar))))
```

To summarize, using a single front-end structure object with arrays as slot values and moving each array separately is much faster than using an array of structures and moving the array into the CM in a single operation.

REFERENCES

See also these related array transfer operations:

array–to–pvar–grid
pvar–to–array **pvar–to–array–grid**

See also the *Lisp operation **pref**, which is used to transfer single values from the CM to the front end.

The *Lisp operation ***setf**, in combination with **pref**, is used to transfer a single value from the front end to the CM.

array–to–pvar–grid [*Defun]

In grid (NEWS) address order, copies values from a front-end array into a pvar.

SYNTAX

array–to–pvar–grid *source–array* **&optional** *dest–pvar*
&key **:array–offset**
:grid–start
:grid–end

ARGUMENTS

source–array	Front-end array. Array from which values are copied. Must have a rank equal to ***number–of–dimensions***.
dest–pvar	Pvar. An allocated pvar into which values are stored. If not supplied, **array–to–pvar–grid** creates a temporary pvar in the current VP set.
:array–offset	Integer list. Set of offsets into *source–array* indicating first value to copy. Default is value of **(make–list *number–of–dimensions* :initial–element 0)**.
:grid–start	Integer list, specifying NEWS (grid) address of processor at which copying starts. Default is value of **(make–list *number–of–dimensions* :initial–element 0)**.
:grid–end	Integer list, specifying NEWS (grid) address of processor at which copying ends. Default is ***current–cm–configuration***.

RETURNED VALUE

dest–pvar	The destination pvar, containing values copied from *source–array*. If *dest–pvar* is supplied, values are copied into it. If not, a temporary pvar is created and returned.

SIDE EFFECTS

The contents of *source–array*, beginning at the element specified by the **:array–offset** argument, are copied into *dest–pvar*. All values of *dest–pvar* specified by the **:grid–start** and **:grid–end** arguments are modified, regardless of the currently selected set. If the *dest–pvar* argument is not supplied, a temporary pvar of the appropriate size is allocated on the stack.

DESCRIPTION

This function copies data from *source–array* to *dest–pvar* in grid (NEWS) address order.

The keyword arguments to **:array–offset**, **:grid–start**, and **:grid–end** must be lists of length ***number–of–dimensions***.

The data from *source–array*, starting with element **:array–offset** as the upper corner, are copied into *dest–pvar*, with **:grid–start** and **:grid–end** specifying the upper and lower corners, respectively. The value returned by **array–to–pvar–grid** is *dest–array*. If *dest–pvar* is unprovided or **nil**, **array–to–pvar–grid** creates a temporary destination pvar. If a destination pvar is created, its value in processors to which **array–to–pvar–grid** did not write is undefined.

It is legal for *source–array* to contain more or fewer elements than can be stored in *dest–pvar*. Extra elements are ignored, and copying an array with fewer elements modifies only a subset of the values of *dest–pvar*.

EXAMPLES

The following expressions select a two-dimensional grid configuration, define a two-dimensional front-end array, and then copy a portion of the array into a pvar on the CM.

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

(defparameter an-array
    (make-array '(5 5)  :element-type 'single-float
                        :initial-element 0.0))

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

The following call transfers the 4 x 4 subarray of **an-array** whose corners are

```
(1 1) (4 1)
(1 4) (4 4)
```

to the 4 x 4 subgrid of **grid-pvar** whose grid-address corners are

```
(2 3) (6 3)
(2 7) (6 7)

(array-to-pvar-grid an-array grid-pvar
   :array-offset '(1 1)
   :grid-start '(2 3))
```

Notice that since the dimensions of **an-array** are (5,5), and copying is specified to begin at (1,1), an array of only (4,4) elements is copied. This in turn means that only a (4,4) subgrid of values is modified in **grid-pvar**.

NOTES

This function is especially useful for copying image data into the Connection Machine. It is much faster than setting pvar elements individually with ***setf** and **pref**.

REFERENCES

See also these related array transfer operations:

array-to-pvar
pvar-to-array **pvar-to-array-grid**

See also the *Lisp operation **pref**, which is used to transfer single values from the CM to the front end.

The *Lisp operation ***setf**, in combination with **pref**, is used to transfer a single value from the front end to the CM.

*array–total–size [*Defun]
array–total–size!! [Function]

Return the total size of each array contained in an array pvar.

SYNTAX

***array–total–size** *array–pvar*
array–total–size!! *array–pvar*

ARGUMENTS

array–pvar Array pvar.

RETURNED VALUE

For ***array–total–size**:

total–size Scalar integer. Total size (product of the lengths of each dimension) of each array contained in *array–pvar*.

For **array–total–size!!**:

size–pvar Temporary integer pvar. In each active processor, contains the total size (product of the lengths of each dimension) of the corresponding value of *array–pvar*.

SIDE EFFECTS

For **array–total–size!!**, the returned pvar is allocated on the stack.

DESCRIPTION

The ***array-total-size** function 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)  <=>
(apply #'* (*array-dimensions array-pvar))
```

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

EXAMPLES

```
(*array-total-size
   (array!! '(2 2) (!! 0) (!! 1) (!! 2) (!! 3))) => 4

(array-total-size!!
   (array!! '(2 2) (!! 0) (!! 1) (!! 2) (!! 3))) <=> (!! 4)
```

NOTES

By definition, an array pvar consists of one array per processor and each array has the same size and shape. Thus, the pvar returned by **array-total-size!!** has the same value in all processors.

```
(array-total-size!! array-pvar)  <=>
(!! (*array-total-size array-pvar))
```

REFERENCES

See also the related array pvar information operators:

***array-dimension**	**array-dimension!!**
***array-dimensions**	**array-dimensions!!**
***array-element-type**	**array-in-bounds-p!!**
***array-rank**	**array-rank!!**
array-row-major-index!!	**sideways-array-p**

ash!! *[Function]*

Performs a parallel arithmetic shift of the supplied pvars.

SYNTAX

ash!! *integer–pvar count–pvar*

ARGUMENTS

integer–pvar	Integer pvar. Value to be shifted.
count–pvar	Integer pvar. Number of bits by which to shift — to the left if positive, to the right if negative.

RETURNED VALUE

shifted–pvar	Temporary integer pvar. Contains in each processor the result of shifting the corresponding value of *integer–pvar* the number of bit positions specified by *count–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The **ash!!** function performs a parallel arithmetic shift operation. It returns a temporary pvar that contains in each active processor the result of shifting the corresponding value of *integer–pvar* the number of bit positions specified by *count–pvar*.

The values in *integer–pvar* are shifted to the left in those processors where *count–pvar* is positive, and to the right where *count–pvar* is negative. In either case, the values from *integer–pvar* are treated as two's-complement integers, and the sign bit is always preserved. In left shifts, zero bits are added from the right; in right shifts, copies of the sign bit are added from the left.

The **ash!!** function provides the same functionality for numeric pvars as the Common Lisp function **ash** provides for numeric scalars.

EXAMPLES

When the values of *count–pvar* are positive, the corresponding values of *integer–pvar* are shifted to the left.

```
(ash!! (!! 2) (!! 0))  <=>  (!! 2)
(ash!! (!! 2) (!! 1))  <=>  (!! 4)
(ash!! (!! 2) (!! 3))  <=>  (!! 16)
(ash!! (!! 2) (!! 9))  <=>  (!! 1024)
```

When the values of *count–pvar* are negative, the corresponding values of *integer–pvar* are shifted to the right.

```
(ash!! (!! 2) (!! -1))     <=>  (!! 1)
(ash!! (!! 2) (!! -2))     <=>  (!! 0)
(ash!! (!! 16) (!! -3))    <=>  (!! 2)
(ash!! (!! 1024) (!! -9))  <=>  (!! 2)
```

The argument *count–pvar* can contain both positive and negative values. For example, if **shift–pvar** contains the values –2, –1, 0, 1, 2, etc., then the pvar returned by

```
(ash!! (!! 4) shift-pvar)
```

contains the values 1, 2, 4, 8, 16, etc.

NOTES

Compiler Note:

This operation will not compile if the bit-length of the *count–pvar* argument is not explicitly declared, because the amount of space allocated by the compiler for an **ash!!** operation depends on the bit-length of this argument.

If the *count–pvar* argument is declared to be of a data type whose length is unspecified, such as **fixnum** in **(ash!! (the (unsigned–byte 4) pvar) (!! (the fixnum x)))**, the compiler will signal an error because there is not enough space to represent the result produced by the largest possible value for this argument. (Specifically, if x had the value 2^{32} then **ash!!** would try to create a pvar roughly 2^{32} bits in length!)

Declarations that explicitly specify the length of the *count–pvar* argument will compile. For example, **(ash!! (the (unsigned–byte 4) pvar) (the (field–pvar 4) x–pvar))** will compile because the result can at most be 19 bits in length (4 bits from the source **pvar,** shifted by up to 15 bits as specified by **x–pvar**).

asin!!, asinh!! [*Function*]

Take the arc sine and arc hyperbolic sine of the supplied pvar.

SYNTAX

asin!! *numeric–pvar*
asinh!! *numeric–pvar*

ARGUMENTS

numeric–pvar Numeric pvar. Pvar for which the arc sine (arc hyperbolic sine) is calculated.

RETURNED VALUE

arc–sine–pvar Temporary numeric pvar. In each active processor, contains the arc sine (arc hyperbolic sine) in radians of the corresponding value of *numeric–pvar.*

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The **asin!!** function calculates the arc sine of *numeric–pvar* in all active processors. It returns a temporary pvar containing in each active processor the arc sine in radians of the corresponding value of *numeric–pvar.* Similarly, the **asinh!!** function calculates the arc hyperbolic sine of *numeric–pvar* in all active processors. The **asin!!** and **asinh!!** functions provide the same functionality for numeric pvars as the Common Lisp functions **asin** and **asinh** provide for numeric scalars.

EXAMPLES

If *numeric–pvar* contains non-complex values, **asin!!** returns the arc sine in each active processor, while **asinh!!** returns the arc hyperbolic sine in each active processor. For example:

```
(asin!! (!! 1.0))            <=>  (!! 1.5707963)
(asinh!! (!! 11.548740))     <=>  (!! 3.1415927)
```

If *numeric–pvar* contains complex values, **asin!!** returns the complex arc sine in each active processor, while **asinh!!** returns the complex arc hyperbolic sine in each active processor:

```
(asin!! (!! #c(1.0 0.0)))           <=>  (!! #c(1.5707963 0.0))
(asinh!! (!! #c(11.548740 0.0)))    <=>  (!! #c(3.1415927 0.0))
```

NOTES

It is an error if *numeric–pvar* contains integer or floating-point values of magnitude greater than 1.0 in any active processor. Complex values with magnitude greater than 1.0 are allowed.

It is an error if *numeric–pvar* contains a non-numeric value in any active processor.

atan!!, atanh!! *[Function]*

Take the arc tangent and arc hyperbolic tangent of the supplied pvar(s).

SYNTAX

atan!! *numeric–pvar* **&optional** *denominator–pvar*
atanh!! *numeric–pvar*

ARGUMENTS

numeric–pvar Numeric pvar. Pvar for which arc tangent (arc hyperbolic tangent) is calculated. Numerator of value if *denominator–pvar* is supplied (for **atan!!** only).

denominator–pvar Numeric pvar. If supplied, denominator of value (for **atan!!** only).

RETURNED VALUE

arc–tangent–pvar Temporary numeric pvar. In each active processor, contains the arc tangent (arc hyperbolic tangent) in radians of the corresponding values in *numeric–pvar* and (if supplied) *denominator–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The **atan!!** function calculates the arc tangent in all active processors.

If only one argument is given, **atan!!** returns a temporary pvar containing in each active processor the arc tangent in radians of the corresponding value of *numeric–pvar.* The argument *numeric–pvar* may contain either real or complex values in this case.

If two arguments are given, the returned pvar contains in each active processor the arc tangent of the quotient of *numeric–pvar* and *denominator–pvar.* The *numeric–pvar* and *denominator–pvar* arguments may not contain complex values in this case. The quadrant of the result is determined by the respective signs of the two arguments. The angle returned in each processor is in standard position, with one side on the x-axis and the other in the same quadrant as the point defined by (*numeric–pvar, denominator–pvar*) in that processor.

The **atanh!!** function calculates the arc hyperbolic tangent of *numeric–pvar* in all active processors. It returns a temporary pvar containing in each active processor the arc hyperbolic tangent in radians of the corresponding value of *numeric–pvar*. The **atanh!!** function provides the same functionality for numeric pvars that the Common Lisp function **atanh** provides for numeric scalars.

The **atan!!** and **atanh!!** functions provide the same functionality for numeric pvars as the Common Lisp functions **atan** and **atanh** provide for numeric scalars.

EXAMPLES

If *numeric–pvar* contains non-complex values, **atan!!** returns the arc tangent in each active processor, while **atanh!!** returns the arc hyperbolic tangent in each active processor:

```
(atan!! (!! 1.0))          <=>  (!! 0.7853982)
(atan!! (!! 3) (!! 4))     <=>  (!! 0.6435011)
(atan!! (!! -3) (!! 4))    <=>  (!! -0.6435011)
(atanh!! (!! .1))          <=>  (!! 0.10033534)
```

If *numeric–pvar* contains complex values, **atan!!** returns the complex arc tangent in each active processor, while **atanh!!** returns the complex arc hyperbolic tangent in each active processor.

```
(atan!! (!! #c(0.27175258 1.08392333))) <=> (!! #c(1.0 0.0))
(atanh!! (!! #c(0.0 0.0)))  <=>  (!! #c(0.0 0.0))
```

NOTES

An error is signalled if *numeric–pvar* and *denominator–pvar* both contain 0 in any active processor, or if either argument contains a non-numeric value in any active processor.

For **atanh!!**: An error is signalled if the argument *numeric–pvar* contains a non-complex value of magnitude greater than or equal to 1 in any active processor.

bit!! [*Function*]

Selects in parallel a bit at a given location in a bit array pvar.

SYNTAX

bit!! *bit–array–pvar* **&rest** *pvar–indices*

ARGUMENTS

bit–array–pvar	Bit array pvar. Array from which bit is selected.
pvar–indices	Integer pvars. Must contain valid subscripts for *bit–array–pvar.* Specifies location of bit to return.

RETURNED VALUE

bit–pvar	Temporary bit pvar. In each processor, contains the bit retrieved from the corresponding array of *bit–array–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function extracts a bit–length pvar from a bit–array pvar.

Note: There is no significant efficiency advantage to using this function in place of **aref!!**; the two are equivalent. Furthermore, you should use **aref!!** instead because **bit!!** will not exist in future versions of *Lisp.

REFERENCES

See also these related bit-array pvar operations:

bit–and!!	**bit–andc2!!**	**bit–not!!**	**bit–orc1!!**	**bit–eqv!!**	**bit–nand!!**
bit–andc1!!	**bit–nor!!**	**bit–orc2!!**	**bit–xor!!**	**bit–ior!!**	**sbit!!**

---

bit–and!!, bit–andc1!!, bit–andc2!!, bit–eqv!!, bit–ior!!, bit–nand!!, bit–nor!!, bit–not!!, bit–orc1!!, bit–orc2!!, bit–xor!! [*Function*]

Perform parallel bitwise logical operations on the supplied bit array pvars.

SYNTAX

bit–not!! *bit–array–pvar–1* **&optional** *destination*

bit–and!!	*bit–array–pvar–1 bit–array–pvar–2* **&optional** *destination*
bit–andc1!!	*bit–array–pvar–1 bit–array–pvar–2* **&optional** *destination*
bit–andc2!!	*bit–array–pvar–1 bit–array–pvar–2* **&optional** *destination*
bit–eqv!!	*bit–array–pvar–1 bit–array–pvar–2* **&optional** *destination*
bit–ior!!	*bit–array–pvar–1 bit–array–pvar–2* **&optional** *destination*
bit–nand!!	*bit–array–pvar–1 bit–array–pvar–2* **&optional** *destination*
bit–nor!!	*bit–array–pvar–1 bit–array–pvar–2* **&optional** *destination*
bit–orc1!!	*bit–array–pvar–1 bit–array–pvar–2* **&optional** *destination*
bit–orc2!!	*bit–array–pvar–1 bit–array–pvar–2* **&optional** *destination*
bit–xor!!	*bit–array–pvar–1 bit–array–pvar–2* **&optional** *destination*

ARGUMENTS

bit–array–pvar–1, bit–array–pvar–2
Bit array pvars. Combined by bitwise logical comparison.

destination Either the value **t**, the value **nil**, or a bit array pvar. Determines where the result is stored. Defaults to **nil**.

RETURNED VALUE

bit–array–result–pvar
Temporary bit array pvar. In each active processor, contains the bitwise logical result. The returned pvar is either a pre-allocated pvar or a temporary pvar, depending on the value of *destination*.

SIDE EFFECTS

If *destination* is **nil** or not supplied, the returned pvar is allocated on the stack. If *destination* is **t**, *bit-array-pvar-1* is destructively modified to contain the result. If *destination* is a bit array pvar, then *destination* is destructively modified to contain the result.

DESCRIPTION

These functions perform logical bitwise operations on the contents of their arguments. The result in each case is a bit array pvar of the same rank and dimensions as the original bit array pvars. It is an error if the arguments are not bit-array pvars of identical rank and dimensionality.

The logical operation performed by each *Lisp function is:

bit-and!!	Bitwise logical AND.
bit-andc1!!	Bitwise logical AND, with *bit-array-pvar-1* complemented.
bit-andc2!!	Bitwise logical AND, with *bit-array-pvar-2* complemented.
bit-eqv!!	Bitwise logical equivalence..
bit-ior!!	Bitwise logical inclusive OR
bit-nand!!	Bitwise logical NAND.
bit-nor!!	Bitwise logical NOR.
bit-not!!	Bitwise logical NOT.
bit-orc1!!	Bitwise logical inclusive OR, with *bit-array-pvar-1* complemented.
bit-orc2!!	Bitwise logical inclusive OR, with *bit-array-pvar-2* complemented.
bit-xor!!	Bitwise logical exclusive OR.

If supplied, the optional *destination* argument must be either **t**, **nil**, or a bit array pvar with the same rank and dimensions as the *bit-array-pvar* arguments. It defaults to **nil**. If *destination* is **nil**, the operation returns a temporary bit array pvar. If *destination* is a bit-array pvar, the result of the operation is destructively stored in that pvar. If *destination* is **t**, the result of the operation is destructively stored in *bit-array-pvar-1*.

EXAMPLES

```
(*defvar bitarr1 (!! #(1 0 1 0)))
(*defvar bitarr2 (!! #(1 1 0 0)))

(bit-and!! bitarr1 bitarr2)       <=>  (!! #(1 0 0 0))

(bit-andc1!! bitarr1 bitarr2)     <=>  (!! #(0 1 0 0))
  <=>    (bit-and (bit-not!! bitarr1) bitarr2)
(bit-andc2!! bitarr1 bitarr2)     <=>  (!! #(0 0 1 0))
  <=>    (bit-and bitarr1 (bit-not!! bitarr2))
(bit-eqv!! bitarr1 bitarr2)       <=>  (!! #(1 0 0 1))

(bit-ior!! bitarr1 bitarr2)       <=>  (!! #(1 1 1 0))

(bit-nand!! bitarr1 bitarr2)      <=>  (!! #(0 1 1 1))

(bit-nor!! bitarr1 bitarr2)       <=>  (!! #(0 0 0 1))

(bit-not!! bitarr1)                         <=>  (!! #(0 1 0 1))

(bit-orc1!! bitarr1 bitarr2)      <=>  (!! #(1 1 0 1))
  <=>    (bit-or!! (bit-not!! bitarr1) bitarr2)
(bit-orc2!! bitarr1 bitarr2)      <=>  (!! #(1 0 1 1))
  <=>    (bit-or!! bitarr1 (bit-not!! bitarr2))
(bit-xor!! bitarr1 bitarr2)       <=>  (!! #(0 1 1 0))
```

REFERENCES

See also these related bit-array pvar operations:

bit!! **sbit!!**

boole!! *[Function]*

Applies boolean operations in parallel to the supplied integer pvars and returns an integer pvar.

SYNTAX

boole!! *op–pvar integer–pvar1 integer–pvar2*

ARGUMENTS

op–pvar Integer pvar. Contains in each processor one of a set of operation constants, described below, that determine the boolean operation performed in that processor.

integer–pvar1, integer–pvar2
Integer pvars. Pvars to which the boolean operation in *op–pvar* is applied.

RETURNED VALUE

integer–result–pvar Temporary integer pvar. In each processor, contains the result of applying the boolean function specified by *op–pvar* to *integer–pvar1* and *integer–pvar2*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The function **boole!!** is the parallel equivalent of the Common Lisp **boole** function.

In each active processor, the logical operation specified by the value of *op–pvar* is performed on the values contained in *integer–pvar1* and *integer–pvar2*.

The following Common Lisp integer 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**

EXAMPLES

A simple call to **boole!!** is

```
(boole!! (!! boole-and) n1 n2)
```

which performs a **boole–and** operation in each processor on **n1** and **n2**. Note that this is equivalent to the expression

```
(logand!! n1 n2)
```

Different logical operations can be performed in different processors. For example, to have **boole–and** execute in all odd processors and **boole–ior** execute in all even processors, use the form

```
(boole!! (if!! (oddp!! (self-address!!))
               (!! boole-and)
               (!! boole-ior))
         n1 n2)
```

REFERENCES

See the definition of the **boole** function in *Common Lisp: The Language.*

booleanp!! *[Function]*

Performs a parallel test for boolean values on the supplied pvar.

SYNTAX

booleanp!! *value–pvar*

ARGUMENTS

value–pvar	Pvar expression. Pvar to be checked for boolean values.

RETURNED VALUE

booleanp–pvar	Temporary boolean pvar. Has the value **t** in each processor in which *value–pvar* contains either **t** or **nil**. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This predicate returns **t** in each processor in which *value–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)**.

EXAMPLES

```
(booleanp!! nil!!)  =>  t!!
```

REFERENCES

See also these related pvar data type predicates:

characterp!!	**complexp!!**	
floatp!!	**front-end-p!!**	**integerp!!**
numberp!!	**string-char-p!!**	**structurep!!**
typep!!		

both–case–p!! *[Function]*

Performs a parallel test for alphabetic characters which have both uppercase and lowercase forms.

SYNTAX

both–case–p!! *character–pvar*

ARGUMENTS

character–pvar	Character pvar. Tested in parallel for dual-case characters.

RETURNED VALUE

both–casep–pvar	Temporary boolean pvar. Contains the value **t** in each active processor where the corresponding value of *character–pvar* is a dual-case character. Contains **nil** in all other processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This predicate tests 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 that may be represented in either upper or lower case, regardless of their current case, **both–case–p!!** returns **t**. Non-Roman fonts, for example, may include alphabetic characters that do not have uppercase or lowercase counterparts.

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

```
(both-case-p!! (!! #\c))  <=>  t!!
(both-case-p!! (!! #\T))  <=>  t!!
(both-case-p!! (!! #\3))  <=>  nil!!
```

byte!! [*Function*]

Creates and returns a byte-specifier pvar suitable as an argument to byte-manipulation functions such as **ldb!!** and **dpb!!**.

SYNTAX

byte!! *size–pvar position–pvar*

ARGUMENTS

size–pvar Integer pvar. Specifies size in bits of byte to be manipulated.

position–pvar Integer pvar. Specified bit position at which byte starts.

RETURNED VALUE

bytespec–pvar Temporary integer pvar. In each active processor, contains a byte-specifier integer formed by combining the values of *size–pvar* and *position–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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.

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!!**.

EXAMPLES

Consider an integer pvar that can be manipulated by one of the byte manipulation functions. If this integer pvar is specifed by a *size-pvar* of **(!! 16)** and a *position-pvar* of **(!! 3)**, we have, in each processor, a 16-bit byte that starts at bit 3 (zero-based). The call to **byte!!** in this instance is

```
(byte!! (!! 16) (!! 3))
```

REFERENCES

See also these related byte manipulation operators:

byte-position!!	**byte-size!!**	
deposit-byte!!	**deposit-field!!**	**dpb!!**
ldb!!	**ldb-test!!**	**load-byte!!**
mask-field!!		

byte–position!!
byte–size!! [*Function*]

Extract the byte position and size component from a byte-specifier pvar.

SYNTAX

byte–position!! *bytespec–pvar*
byte–size!! *bytespec–pvar*

ARGUMENTS

bytespec–pvar Byte-specifier pvar, as returned by the function **byte!!**.

RETURNED VALUE

position–pvar Temporary integer pvar. In each active processor, contains the extracted component of *bytespec–pvar.*

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The functions **byte–position!!** and **byte–size!!** each take a byte specifier pvar, created by calling **byte!!**, as their argument.The integer pvar returned is a copy of the *position–pvar* or *size–pvar* originally given as an argument to **byte!!**. Thus:

```
(byte-position!!   (byte!! size pos))  <=>  (!! pos)
(byte-size!!       (byte!! size pos))  <=>  (!! size)
```

EXAMPLES

```
(byte-position!!   (byte!! (!! 16) (!! 3)))  <=>  (!! 3)
(byte-size!!       (byte!! (!! 16) (!! 3)))  <=>  (!! 16)
```

REFERENCES

See also these related byte manipulation operators:

byte!!	**deposit–byte!!**	**deposit–field!!**	**dpb!!**
ldb!!	**ldb–test!!**	**load–byte!!**	**mask–field!!**

*case, case!! [*Macro*]

Evaluates *Lisp forms with the currently selected set bound according to the value of a pvar expression.

Returns a pvar obtained by evaluating *Lisp forms with the currently selected set bound according to the value of a pvar expression.

SYNTAX

***case/case!!** *value–expression* (*key–expression–1* **&rest** *body–forms–1*)
(*key–expression–2* **&rest** *body–forms–2*)
...
(*key–expression–n* **&rest** *body–forms–n*)

ARGUMENTS

value–expression Pvar expression. Value to compare against *key–expression–n* in each clause.

key–expression–n Scalar expression. Evaluated, compared with *value–expression.* Selects processors in which to perform the corresponding *body–forms*. May also be a list of such expressions, in which case each expression is compared with *value–expression.*

body–forms–n *Lisp forms. These forms are evaluated with the currently selected set restricted to those processors in which *value–expression* is **eql!!** to **(!!** *key–expression–n* **)**.

RETURNED VALUE

For ***case**:

nil Evaluated for side effect only.

For **case!!**:

case–value–pvar
Temporary pvar. In each active processor, contains the value returned by *body–forms–n* if and only if *value–expression* is **eql** to *key–expression–n*.

SIDE EFFECTS

For ***case**:

None aside from those of the individual *body-forms*.

For **case!!**:

The returned pvar is allocated on the stack.

DESCRIPTION

The ***case** and **case!!** macros are parallel equivalents of the Common Lisp **case** operation. The two operators each select groups of processors to execute different portions of *Lisp code. Unlike **case**, however, ***case** and **case!!** evaluate all clauses.

The main difference between ***case** and **case!!** is that ***case** is used only for the side effects of its body forms, while **case!!** also constructs and returns a value-pvar that contains the value returned by its *body-forms*.

EXAMPLES

When the following forms are evaluated,

```
(*defvar result (!! 1))
(*case (mod!! (self-address!!) (!! 4))
   (0      (*set result (!! 0)))
   ((1 2)  (*set result (self-address!!)))
   (otherwise (*set result (!! -1))))
```

result is bound to a pvar with the values 0, 1, 2, –1, 0, 5, 6, –1, etc.

Similarly, when

```
(case!! (mod!! (self-address!!) (!! 4))
   (0      (!! 0))
   ((1 2)  (self-address!!))
   (otherwise (!! -1)))
```

is executed, the returned pvar contains the values 0, 1, 2, –1, 0, 5, 6, –1, etc.

NOTES

Usage Notes:

It is an error for two ***case** or **case!!** clauses to contain the same *key–expression*. If two **case!!** clauses contain the same key, the returned pvar contains the values returned by the body forms in the first of the clauses.

Forms such as **throw, return, return–from,** and **go** may be used to exit a block or looping construct from within a processor selection operator such as ***case** or **case!!**. However, doing so will leave the currently selected set in the state it was in at the time the non-local exit form is executed. To avoid this, use the *Lisp macro **with–css–saved**. See the dictionary entry for **with–css–saved** for more information.

Performance Note:

Currently, ***case** and **case!!** clauses execute serially, in the order in which they are supplied. At any given time, therefore, the number of processors active within a ***case** or **case!!** clause is a subset of the currently selected set at the time the ***case** or **case!!** form was entered. Providing a large number of clauses therefore can result in inefficient processor usage.

REFERENCES

See also the related operators

***all**	***cond**	**cond!!**	***ecase**	**ecase!!**
***if**	**if!!**	***unless**	***when**	**with–css–saved**

ceiling!! [*Function*]

Performs a parallel ceiling operation on the supplied pvar(s).

SYNTAX

ceiling!! *numeric–pvar* **&optional** *divisor–numeric–pvar*

ARGUMENTS

numeric–pvar Non-complex numeric pvar. Value for which the ceiling is calculated.

divisor–numeric–pvar
Non-complex numeric pvar. If supplied, *numeric–pvar* is divided by *divisor–numeric–pvar* before the ceiling is taken.

RETURNED VALUE

ceiling–pvar Temporary integer pvar. In each active processor, contains the ceiling of *numeric–pvar*, divided by *divisor–numeric–pvar* if supplied.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This is the parallel equivalent of the Common Lisp function **ceiling**, except that only one value—the quotient of the division—is computed and returned.

EXAMPLES

```
(ceiling!! (!! 4.5))  <=>  (!! 5)
```

REFERENCES

See also these related rounding operations:

floor!! **round!!** **truncate!!**

See also these related floating-point rounding operations:

fceiling!! **ffloor!!** **fround!!** **ftruncate!!**

char=!!, char/=!!, char<!!, char<=!!, char>!!,char>=!! [*Function*]

Perform a case-sensitive parallel comparison of the supplied character pvars.

SYNTAX

char=!!	*character-pvar*	**&rest**	*character-pvars*
char/=!!	*character-pvar*	**&rest**	*character-pvars*
char<!!	*character-pvar*	**&rest**	*character-pvars*
char<=!!	*character-pvar*	**&rest**	*character-pvars*
char>!!	*character-pvar*	**&rest**	*character-pvars*
char>=!!	*character-pvar*	**&rest**	*character-pvars*

ARGUMENTS

character-pvar, character-pvars
Character pvars. Compared in parallel.

RETURNED VALUE

char-comparison-pvar
Temporary boolean pvar. Contains **t** in each active processor where all of the supplied *character-pvar* arguments satisfy the character comparison. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The tests performed by these operations are as follows:

char=!!	case-sensitive ASCII value equality
char/=!!	case-sensitive ASCII value inequality
char<!!	case-sensitive strictly increasing ASCII ordering

char<=!!	case-sensitive nondecreasing ASCII ordering
char>!!	case-sensitive strictly decreasing ASCII ordering
char>=!!	case-sensitive nonincreasing ASCII ordering

EXAMPLES

```
(char=!! (!! #\c) (!! #\c))  <=>  t!!
(char=!! (!! #\c) (!! #\C))  <=>  nil!!
(char=!! (!! #\c) (!! #\3))  <=>  nil!!
(char=!! (!! #\c) (!! #\z))  <=>  nil!!
(char=!! (!! #\c) (!! #\c) (!! #\c))  <=>  t!!
(char=!! (!! #\c) (!! #\c) (!! #\C))  <=>  nil!!
(char=!! (!! #\c) (!! #\Z) (!! #\C))  <=>  nil!!

(char/=!! (!! #\c) (!! #\c))  <=>  nil!!
(char/=!! (!! #\c) (!! #\C))  <=>  t!!
(char/=!! (!! #\c) (!! #\3))  <=>  t!!
(char/=!! (!! #\c) (!! #\z))  <=>  t!!
(char/=!! (!! #\c) (!! #\c) (!! #\c))  <=>  nil!!
(char/=!! (!! #\c) (!! #\c) (!! #\C))  <=>  nil!!
(char/=!! (!! #\c) (!! #\Z) (!! #\C))  <=>  t!!

(char<!! (!! #\c) (!! #\c))  <=>  nil!!
(char<!! (!! #\c) (!! #\C))  <=>  nil!!
(char<!! (!! #\c) (!! #\3))  <=>  nil!!
(char<!! (!! #\c) (!! #\z))  <=>  t!!
(char<!! (!! #\A) (!! #\B) (!! #\Z) ) <=> t!!

(char<=!! (!! #\c) (!! #\c))  <=>  t!!
(char<=!! (!! #\c) (!! #\C))  <=>  nil!!
(char<=!! (!! #\c) (!! #\3))  <=>  nil!!
(char<=!! (!! #\c) (!! #\z))  <=>  t!!
(char<=!! (!! #\1) (!! #\5) (!! #\5) ) <=> t!!

(char>!! (!! #\c) (!! #\c))  <=>  nil!!
(char>!! (!! #\c) (!! #\C))  <=>  t!!
(char>!! (!! #\c) (!! #\3))  <=>  t!!
(char>!! (!! #\c) (!! #\z))  <=>  nil!!
(char>!! (!! #\z) (!! #\j) (!! #\a) ) <=> t!!

(char>=!! (!! #\c) (!! #\c))  <=>  t!!
(char>=!! (!! #\c) (!! #\C))  <=>  t!!
(char>=!! (!! #\c) (!! #\3))  <=>  t!!
(char>=!! (!! #\c) (!! #\z))  <=>  nil!!
(char>=!! (!! #\5) (!! #\1) (!! #\1) ) <=> t!!
```

character!!

[Function]

Coerces the supplied pvar into a character pvar.

SYNTAX

character!! *char–or–int–pvar*

ARGUMENTS

char–or–int–pvar Pvar containing only integer or character values. Pvar to be coerced into a character pvar. Must be a pvar of type character, string-char, integer, or a general pvar containing only elements of these types.

RETURNED VALUE

char–pvar Temporary character pvar. In each active processor, contains the character equivalent of the corresponding value of *char–or–int–pvar*, or the value **nil** if coercion could not be performed.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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.**

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

REFERENCES

See also the related character pvar constructor **make–char!!**.

See also the related character pvar attribute operators:

char–bit!!	**char–bits!!**	**char–code!!**
char–font!!	**initialize–character**	**set–char–bit!!**

characterp!! [*Function*]

Performs a parallel test for character values on the supplied pvar.

SYNTAX

characterp!! *pvar*

ARGUMENTS

pvar	Pvar expression. Pvar to be tested for character values.

RETURNED VALUE

characterp–pvar	Temporary boolean pvar. Contains the value **t** in each active processor where *pvar* contains a character value. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function returns **t** in all active processors where the supplied *pvar* contains character data and **nil** in all other active processors.

EXAMPLES

```
(characterp!! (!! #\c))  <=>  t!!
(characterp!! (!! 0))    <=>  nil!!
```

REFERENCES

See also these related pvar data type predicates:

booleanp!!	**complexp!!**	
floatp!!	**front-end-p!!**	**integerp!!**
numberp!!	**string-char-p!!**	**structurep!!**
typep!!		

char–bit!!

[Function]

Tests the state of a single flag bit of the supplied character pvar.

SYNTAX

char–bit!! *character–pvar bit–name–pvar*

ARGUMENTS

character–pvar — Character pvar. Pvar for which bit selected by *bit–name–pvar* is tested.

bit–name–pvar — Integer pvar. Selects bit to be tested in each active processor. Must contain integers in the range 0 to 3 inclusive.

RETURNED VALUE

flag–state–pvar — Temporary boolean pvar. Contains the value **t** in each active processor where the flag bit named by *bit–name–pvar* in *character–pvar* is set. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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!!

(char-bit!! char-pvar (!! x))  <=>
(logbitp!! (!! x) (char-bits!! char-pvar))
```

REFERENCES

See also the related character pvar attribute operators:

char–bits!! **char–code!!**
char–font!! **initialize–character** **set–char–bit!!**

char–bits!! [*Function*]

Extracts in parallel the bits attribute of a character pvar.

SYNTAX

char–bits!! *character–pvar*

ARGUMENTS

character–pvar	Character pvar. Pvar from which to extract the bits attribute.

RETURNED VALUE

char–bits–pvar	Temporary integer pvar. In each active processor, contains an integer representing the bits attribute of the corresponding value of *character–pvar.*

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function returns a pvar that contains the bits attribute 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)
```

REFERENCES

See also the related character pvar attribute operators:

char-bit!!	**char-code!!**	
char-font!!	**initialize-character**	**set-char-bit!!**

char–code!! *[Function]*

Extracts in parallel the code attribute of a character pvar.

SYNTAX

char–code!! *character–pvar*

ARGUMENTS

character–pvar Character pvar. Pvar from which to extract the code attribute.

RETURNED VALUE

char–code–pvar Temporary integer pvar. In each active processor, contains an integer representing the code attribute of the corresponding value of *character–pvar.*

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function returns a pvar that contains the code attribute 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.

```
(char-code!! #\A)  <=>  (!! 65)
```

REFERENCES

See also the related character pvar attribute operators:

char–bit!! **char–bits!!**
char–font!! **initialize–character** **set–char–bit!!**

See also the related character/integer pvar conversion operators:

char-int!!	**code-char!!**	**digit-char!!**
int-char!!		

char–downcase!! [*Function*]

Converts uppercase alphabetic characters in the supplied pvar to lowercase.

SYNTAX

char–downcase!! *character–pvar*

ARGUMENTS

character–pvar Character pvar. Pvar containing characters to be converted. Must be a pvar of type character or string-char, or a general pvar containing only elements of these types.

RETURNED VALUE

downcase–pvar Temporary character pvar. In each active processor, contains a copy of the corresponding value of *character–pvar,* with uppercase characters converted to their lowercase equivalents.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function attempts to convert the case of each character element of *character–pvar*. The returned value 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 affected by case conversion. Thus, characters with non-zero bit-field values are not changed.

EXAMPLES

```
(char-downcase!! (!! #\C))  <=>  (!! #\c)
(char-downcase!! (!! #\c))  <=>  (!! #\c)
(char-downcase!! (!! #\3))  <=>  (!! #\3)
```

char–equal!! *[Function]*

Performs a case-insensitive parallel comparison of the supplied character pvars for equality.

SYNTAX

char–equal!! *character–pvar* **&rest** *character–pvars*

ARGUMENTS

character–pvar	Character pvar. Compared in parallel for case-insensitive equality.
character–pvars	Character pvars. Compared in parallel for case-insensitive equality.

RETURNED VALUE

char–equal–pvar	Temporary boolean pvar. Contains the value **t** in each active processor where all of the supplied *character–pvar* arguments contain the same character, regardless of case. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function makes a case-insensitive comparison between the character element of *character–pvar* in each processor and the character elements of each of the *character–pvars* in the same processor. Differences in case, bit, and font attributes are ignored.

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

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

EXAMPLES

```
(char-equal!! (!! #\c) (!! #\c))  <=>  t!!
(char-equal!! (!! #\c) (!! #\C))  <=>  t!!
(char-equal!! (!! #\c) (!! #\3))  <=>  nil!!
(char-equal!! (!! #\c) (!! #\z))  <=>  nil!!
```

char–flipcase!! [*Function*]

In the supplied pvar, converts uppercase characters to lowercase, and vice-versa.

SYNTAX

char–flipcase!! *character–pvar*

ARGUMENTS

character–pvar	Character pvar. Pvar containing characters to be converted. Must be a pvar of type character or string-char, or a general pvar containing only elements of these types.

RETURNED VALUE

downcase–pvar	Temporary character pvar. In each active processor, contains a copy of the corresponding value of *character–pvar,* with uppercase characters converted to lowercase, and lowercase characters converted to uppercase.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function attempts to invert the case of each character element of *character–pvar.* The return value 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 affected by case conversion. Thus, characters with non-zero bit field values are not changed.

EXAMPLES

```
(char-flipcase!! (!! #\C))  <=>  (!! #\c)
(char-flipcase!! (!! #\c))  <=>  (!! #\C)
(char-flipcase!! (!! #\3))  <=>  (!! #\3)
```

char–font!! [*Function*]

Extracts in parallel the font attribute of a character pvar.

SYNTAX

char–font!! *character–pvar*

STNTAX

ARGUMENTS

character–pvar	Character pvar. Pvar from which to extract the font attribute. Must be a character pvar, a string-char pvar, or a general pvar containing only character or string-char elements.

RETURNED VALUE

font–pvar	Temporary integer pvar. In each active processor, contains a integer representing the font attribute of the corresponding value of *character–pvar.*

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function returns a pvar that contains the font attributes of each character element of *character–pvar*.

NOTES

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

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

REFERENCES

For a discussion of Common Lisp character attributes (code, bits, and font), see *Common Lisp: The Language*, Chapter 13.

See also the related character pvar attribute operators:

char–bit!!	**char–bits!!**	**char–code!!**
initialize–character	**set–char–bit!!**	

char–greaterp!! *[Function]*

Performs a case-insensitive parallel comparison of the supplied character pvars for decreasing order.

SYNTAX

char–greaterp!! *character–pvar* **&rest** *character–pvars*

ARGUMENTS

character–pvar Character pvar. Compared in parallel for case-insensitive decreasing order.

character–pvars Character pvars. Compared in parallel for case-insensitive decreasing order.

RETURNED VALUE

char–greaterp–pvar

Temporary boolean pvar. Contains the value **t** in each active processor where the supplied *character–pvar* arguments are in case-insensitive decreasing order. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function makes a case-insensitive comparison between the character element of *character–pvar* in each processor and the character elements of each of the *character–pvars* in the same processor. Differences in case, bit, and font attributes are ignored.

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

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

EXAMPLES

```
(char-greaterp!! (!! #\Z) (!! #\N) (!! #\A))  <=>  t!!
(char-greaterp!! (!! #\Z) (!! #\z))  <=>  nil!!
```

char-int!! [*Function*]

Converts the supplied character pvar into an integer pvar.

SYNTAX

char-int!! *character-pvar*

ARGUMENTS

character-pvar Character pvar. Pvar to be converted. Must be a pvar of type character or string-char, or a general pvar containing only elements of these types.

RETURNED VALUE

integer-pvar Temporary integer pvar. In each active processor, contains the integer value of the character in *character-pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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*.

EXAMPLES

```
(char-int!! (!! #\A))  <=>  (!! 65)
```

NOTES

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.

REFERENCES

See also the related character/integer pvar conversion operators:

char-code!! **code-char!!** **digit-char!!**
int-char!!

char–lessp!! [*Function*]

Performs a case-insensitive parallel comparison of the supplied character pvars for increasing order.

SYNTAX

char–lessp!! *character–pvar* **&rest** *character–pvars*

ARGUMENTS

character–pvar	Character pvar. Compared in parallel for case-insensitive increasing order.
character–pvars	Character pvars. Compared in parallel for case-insensitive increasing order.

RETURNED VALUE

char–greaterp–pvar

Temporary boolean pvar. Contains the value **t** in each active processor where the supplied *character–pvar* arguments are in case-insensitive increasing order. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function makes case-insensitive comparisons between the character element of *character–pvar* in each processor and the character elements of each of the *character–pvars* in the same processor. Differences in case, bit, and font attributes are ignored.

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

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

EXAMPLES

```
(char-lessp!! (!! #\A) (!! #\N) (!! #\Z))  <=>  t!!
(char-lessp!! (!! #\Z) (!! #\z))  <=>  nil!!
```

char-not-equal!! *[Function]*

Performs a case-insensitive parallel comparison of the supplied character pvars for inequality.

SYNTAX

char-not-equal!! *character-pvar* **&rest** *character-pvars*

ARGUMENTS

character-pvar	Character pvar. Compared in parallel for case-insensitive inequality.
character-pvars	Character pvars. Compared in parallel for case-insensitive inequality.

RETURNED VALUE

char-equal-pvar	Temporary boolean pvar. Contains the value **t** in each active processor where all of the supplied *character-pvar* arguments contain different characters, case-insensitive. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function makes case-insensitive comparisons between the character element of *character-pvar* in each processor and the character elements of each of the *character-pvars* in the same processor. Differences in case, bit, and font attributes are ignored.

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

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

EXAMPLES

```
(char-not-equal!! (!! #\c) (!! #\c))  <=>  nil!!
(char-not-equal!! (!! #\c) (!! #\C))  <=>  nil!!
(char-not-equal!! (!! #\c) (!! #\3))  <=>  t!!
(char-not-equal!! (!! #\c) (!! #\z))  <=>  t!!
```

char-not-greaterp!! *[Function]*

Performs a case-insensitive parallel comparison of the supplied character pvars for nondecreasing order.

SYNTAX

char-not-greaterp!! *character-pvar* **&rest** *character-pvars*

ARGUMENTS

character-pvar Character pvar. Compared in parallel for case-insensitive nondecreasing order.

character-pvars Character pvars. Compared in parallel for case-insensitive nondecreasing order.

RETURNED VALUE

char-not-greaterp-pvar

Temporary boolean pvar. Contains the value **t** in each active processor where the supplied *character-pvar* arguments are in case-insensitive nondecreasing order. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function makes case-insensitive comparisons between the character element of *character-pvar* in each processor and the character elements of each of the *character-pvars* in the same processor. Differences in case, bit, and font attributes are ignored.

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

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

EXAMPLES

```
(char-not-greaterp!! (!! #\Z) (!! #\N) (!! #\A))  <=>  nil!!
(char-not-greaterp!! (!! #\Z) (!! #\z))  <=>  t!!
```

char–not–lessp!! *[Function]*

Performs a case-insensitive parallel comparison of the supplied character pvars for nonincreasing order.

SYNTAX

char–not–lessp!! *character–pvar* **&rest** *character–pvars*

ARGUMENTS

character–pvar Character pvar. Compared in parallel for case-insensitive nonincreasing order.

character–pvars Character pvars. Compared in parallel for case-insensitive nonincreasing order.

RETURNED VALUE

char–not–greaterp–pvar

Temporary boolean pvar. Contains the value **t** in each active processor where the supplied *character–pvar* arguments are in case-insensitive nonincreasing order. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function makes case-insensitive comparisons between the character element of *character–pvar* in each processor and the character elements of each of the *character–pvars* in the same processor. Differences in case, bit, and font attributes are ignored.

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

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

EXAMPLES

```
(char-not-lessp!! (!! #\A) (!! #\N) (!! #\Z))  <=>  t!!
(char-not-lessp!! (!! #\Z) (!! #\z))  <=>  nil!!
```

char–upcase!! *[Function]*

Converts lowercase alphabetic characters in the supplied pvar to uppercase.

SYNTAX

char–upcase!! *character–pvar*

ARGUMENTS

character–pvar Character pvar. Pvar containing characters to be converted. Must be a pvar of type character or string-char, or a general pvar containing only elements of these types.

RETURNED VALUE

upcase–pvar Temporary character pvar. In each active processor, contains a copy of the corresponding value of *character–pvar,* with lowercase characters converted into their uppercase equivalents.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function attempts to convert the case of each character element of *character–pvar*. The return value 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 affected by case conversion. Thus, characters with non-zero bit field values are not changed.

EXAMPLES

```
(char-upcase!! (!! #\C))  <=>  (!! #\C)
(char-upcase!! (!! #\c))  <=>  (!! #\C)
(char-upcase!! (!! #\3))  <=>  (!! #\3)
```

cis!! [*Function*]

Performs a parallel conversion of phase angles into unit-length complex numbers.

SYNTAX

cis!! *numeric–pvar*

ARGUMENTS

numeric–pvar	Non-complex numeric pvar. Phase angle in radians to convert to a complex number.

RETURNED VALUE

cos–i–sin–pvar	Temporary complex pvar. In each active processor, contains a unit-length complex number with a phase angle equal to the corresponding value of *numeric–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function is the parallel equivalent of the Common Lisp function **cis**. It returns a temporary complex pvar whose value in each processor is a complex number of unit length, whose phase is the value of the corresponding value of *numeric–pvar*.

```
(cis!! (!! 3.1415927))  <=>  (!! #c(-1.0 2.3841858e-7))
```

Another way to view this function is as returning the position on a unit circle, centered on the complex plane, that corresponds to the angle stored in each processor of a pvar (see Figure 1).

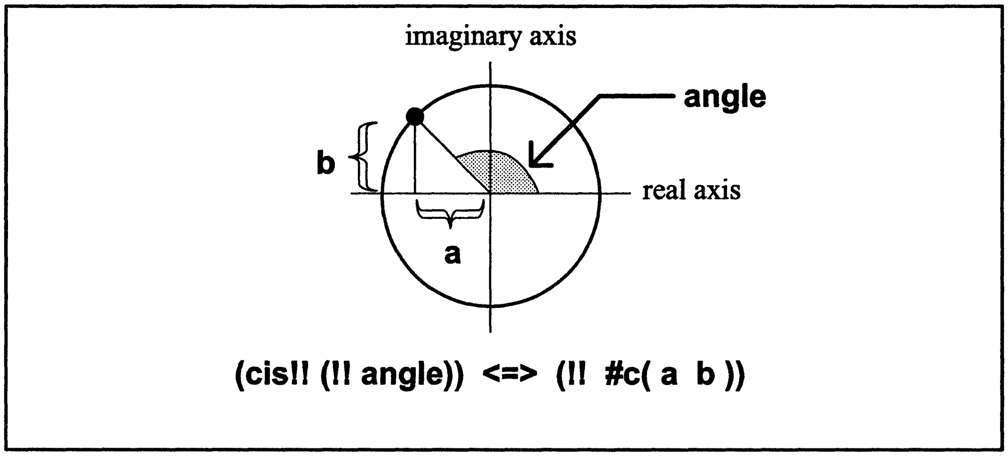

Figure 1. The function **cis!!** calculates positions on a unit circle centered in the complex plane.

REFERENCES

See also these related complex pvar operators:

abs!!	**complex!!**	
conjugate!!	**imagpart!!**	**phase!!**
realpart!!		

code–char!! [*Function*]

Converts numeric pvar of character codes to a character pvar with the supplied attributes.

SYNTAX

code–char!! *code–pvar* **&optional** *bits–pvar font–pvar*

ARGUMENTS

code–pvar	Non-negative integer pvar. Code attribute of character pvar.
bits–pvar	Non-negative integer pvar. Bits attribute of character pvar.
font–pvar	Non-negative integer pvar. Font attribute of character pvar.

RETURNED VALUE

char–pvar	Temporary character pvar. In each active processor, contains a character with the code, bits, and font attributes specified by the corresponding values of *code–pvar*, *bits–pvar*, and *font–pvar*. Contains **nil** in processors where the specified character can not be constructed.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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)**.

EXAMPLES

```
(code-char!! (!! 65))  <=>  (!! #\A)
```

REFERENCES

For a discussion of Common Lisp character attributes (code, bits, and font), see *Common Lisp: The Language*, Chapter 13.

See also the related character pvar attribute operators:

char–bit!!	**char–bits!!**	**char–code!!**
char–font!!	**initialize–character**	**set–char–bit!!**

See also the related character/integer pvar conversion operators:

char–code!!	**char–int!!**	**digit–char!!**
int–char!!		

coerce!! [*Function*]

Performs a parallel type coercion on the supplied pvar.

SYNTAX

coerce!! *pvar type–spec*

ARGUMENTS

pvar	Pvar expression. Pvar containing values to be coerced.
type–spec	Type specifier. Must specify a valid *Lisp pvar type.

RETURNED VALUE

coerced–pvar	Temporary pvar. Result of coercing *pvar* to the pvar type specified by *type–spec*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The **coerce!!** function is the parallel equivalent of the Common Lisp **coerce** function.This function attempts to convert *pvar* to the type indicated by *type–spec*. If this is possible, the result is returned as a new pvar allocated on the *Lisp stack. If *pvar* is already of type *type–spec*, a copy of *pvar* is returned. If the specified conversion is not possible, an error is signaled.

Important: in many simple cases, type conversion is performed automatically. For example, arithmetic operations such as **+!!** and pvar copying functions such as ***set** automatically coerce their arguments according to the rules of *Lisp type coercion. It is only necessary to explicitly **coerce!!** a pvar in special cases, such as converting a numeric pvar to a larger bit size or altering the element type of an array pvar.

EXAMPLES

It is not generally possible to convert a given pvar to any data type; 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 byte size. For instance, a pvar of type **(pvar (unsigned-byte 8))** may be coerced to **(pvar (signed-byte 16))**

  ```
  (*proclaim '(type (pvar (unsigned-byte 8)) data-8))
  (*defvar data-8 (random!! (!! 20)))
  (*proclaim '(type (pvar (unsigned-byte 16)) data-16))
  (*defvar data-16)
  (*set data-16
     (coerce!! data-8 '(pvar (signed-byte 16))))
  ```

  Conversions to smaller byte sizes are also legal. For example, a pvar of type **(pvar (unsigned-byte 8))** may be coerced to **(pvar (unsigned-byte 4))**

  ```
  (*proclaim '(type (pvar (unsigned-byte 4)) data-4))
  (*defvar data-4 (random!! (!! 4)))

  (*set data-4 (coerce!! data-8 '(pvar (signed-byte 4))))
  ```

- 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)**

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

  (*set data-sf (coerce!! data-16 '(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)**

  ```
  (*proclaim '(type double-float-pvar data-df))
  (*defvar data-df)

  (*set data-df (coerce!! data-sf '(pvar double-float)))
  ```

- An integer pvar or a floating-point 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 **double-float**

```
(*proclaim '(type double-complex-pvar data-df-complex))
(*defvar data-df-complex)

(*set data-df-complex
      (coerce!! data-sf 'double-complex-pvar))
```

- 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**

```
(*proclaim '(type single-complex-pvar data-sf-complex))
(*defvar data-sf-complex (complex!! (!! 1.0) (!! -1.0)))

(*set data-df-complex
      (coerce!! data-sf-complex '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!!**

```
(*proclaim '(type character-pvar data-char))
(*defvar data-char)

(*set data-char
   (coerce!! (random!! (!! 65)) 'character-pvar))
```

- A string-char array pvar of length 1 may be converted to a character pvar.

```
(*proclaim '(type (pvar (array string-char (1)))
                 data-string-char))
(*defvar data-string-char (!! "C"))
(*set data-char
      (coerce!! data-string-char 'character-pvar))
```

- Any pvar, except an array or a structure pvar. may be converted to a general pvar.

```
(*proclaim '(type (pvar front-end) data-front-end))
(*defvar data-front-end (front-end!! 'commander))
(*proclaim '(type (pvar t) data-general))
(*defvar data-general)

(*set data-general (coerce!! data-front-end '(pvar t)))
```

- 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.

```
(*proclaim '(type (pvar (array single-float (20)))
                  data-array-sf))
(*defvar data-array-sf
         (make-array!! '(20)
               :initial-element (random!! (!! 2.0))
               :element-type 'single-float))
(*proclaim '(type (pvar (array double-float (20)))
                 data-array-df))
(*defvar data-array-df)

(*set data-array-df (coerce!! data-array-sf
                      '(pvar (array double-float (20)))))
```

NOTES

Explicit type conversion functions may be used in place of **coerce!!**.

Examples of *Lisp functions in this category are:

ceiling!!	**character!!**	**complex!!**
float!!	**floor!!**	**round!!**
truncate!!		

REFERENCES

See also the related *Lisp declaration operators:

***locally**	***proclaim**	**unproclaim**

See also the related type translation function **taken-as!!**.

*cold–boot [*Macro*]

Initializes *Lisp, resets the Connection Machine hardware, and defines the current machine configuration and default VP set.

SYNTAX

***cold–boot &key :safety**
:initial–dimensions
:initial–geometry–definition
:undefine–all
:physical–size
&allow–other–keys *attach–keywords*

ARGUMENTS

:safety An integer between 0 and 3, inclusive. Specifies a value for the *Lisp variable ***interpreter–safety***. Defaults to 3, the highest safety level.

:initial–dimensions
A list of integers, each of which must be a power of 2. Defines the dimensions of the ***default–vp–set***. Defaults to a two-dimensional grid with a VP ratio of 1.

:initial–geometry–definition
Geometry object, as returned by **create–geometry**. May be supplied instead of an **:initial–dimensions** argument to define the geometry of the ***default–vp–set***.

:undefine–all Boolean value. Determines whether currently defined VP sets and permanent pvars are reallocated. Defaults to **nil**, indicating that VP sets and permanent pvars should be reallocated.

:physical–size Physical-size argument (either a number of processors or a keyword selecting a machine size or sequencer). Passed to **cm:attach** if ***cold–boot** calls it to attach to a CM

attach–keywords Other keywords. These extra keyword arguments are passed along to **cm:attach** if ***cold–boot** calls it to attach to a CM.

RETURNED VALUE

physical-size The value of ***minimum-size-for-vp-set***, equal to the number of physical processors attached.

dimensions The value of ***current-cm-configuration***, a list of integers defining the geometry of the ***default-vp-set***.

SIDE EFFECTS

Initializes *Lisp and Connection Machine hardware. If **:undefine-all** is **nil**, reallocates permanent pvars and VP sets. Attempts to attach to CM hardware if not already attached.

DESCRIPTION

The ***cold-boot** macro initializes the *Lisp system and resets the Connection Machine hardware. It should be called immediately after loading in the *Lisp software and attaching to a Connection Machine, and before executing *Lisp code that does anything other than defining pvars (with ***defvar**) and defining VP sets. The ***cold-boot** macro may also be called from top level at any time to change the processor configuration of the Connection Machine.

In general, ***cold-boot** should be called only from top level or at the very beginning of the main function of a program. It should never be called at any other point in a program, because it resets the entire state of *Lisp and the Connection Machine.

The **:safety** keyword argument specifies a value for the *Lisp global variable ***interpreter-safety***. See the description of ***interpreter-safety*** in Chapter 2, "*Lisp Global Variables", for a description of interpreter safety levels.

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

The **:initial-dimensions** keyword argument specifies the dimensions of the Connection Machine processor configuration. For example, an **:initial-dimensions** argument of **(32 16 64)** specifies a three-dimensional processor configuration with dimensions 32 x 16 x 64. The dimensions must be powers of 2. The product of the dimensions must be either equal to the number of physical processors attached, or equal to a power of two multiple of the number of attached processors.

The **:initial–geometry–definition** allows the use of a geometry object to specify the processor configuration. Supplying a geometry object instead of a list of dimensions permits greater control over the routing pattern and processor address mapping of the default VP set. See the definition of **create–geometry** for more information about creating and using geometry objects.

If neither the **:initial–dimensions** nor the **:initial–geometry–definition** arguments is supplied, the dimensions default to the same configuration as that used in the previous call to ***cold–boot**. If there was no previous call, the default is a two-dimensional grid with a VP ratio of 1.

The **:undefine–all** keyword determines whether all permanent VP sets and permanent pvars are reallocated. If **:undefine–all** is **nil**, the default, all permanent VP sets and permanent pvars are automatically reallocated. If this argument is non-**nil**, ***cold–boot** deallocates and destroys all permanent pvars and all VP sets with the exception of the ***default–vp–set*** and its associated geometry object.

In detail, calling ***cold–boot** performs the following operations in sequence:

- evaluates in order the forms on the ***before–*cold–boot–initializations*** list
- deallocates all previously defined pvars, *including* permanent pvars
- deallocates all previously defined VP sets
- attempts to attach to Connection Machine hardware—if not already attached—and calls the Paris function **cm:cold–boot** if successful
- sets the value of the variable ***interpreter–safety***
- instantiates the VP set bound to ***default–vp–set*** with a geometry based on the values of the **:initial–dimensions** and **:initial–geometry–definition** arguments
- if **:undefine–all** is **nil**, redefines all permanent VP sets in an arbitrary order, and instantiates all fixed-size VP sets
- if **:undefine–all** is **nil**, reallocates and reinitializes, using ***defvar**, permanent pvars that belong to instantiated VP sets
- selects the VP set ***default–vp–set***, making it the ***current–vp–set***
- evaluates in order the forms on the ***after–*cold–boot–initializations*** list

EXAMPLES

Here are some sample calls to ***cold–boot**, defining various configurations of processors.

```
(*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))
```

Here is a sample call to ***cold–boot** using a geometry object to define the processor configuration.

```
(defvar my-geometry
   (create-geometry :dimensions '(2 32 2) :weights '(2 1 3)))

(*cold-boot :initial-geometry-definition my-geometry)
```

The next two examples assume that a Connection Machine with 8K processors is attached, and that no previous call to ***cold–boot** has been made. The first example defines a configuration with a VP ratio of 1, i.e., one virtual processor for each physical processor. Because no dimensions are supplied, a 2-dimensional grid of processors is defined, with dimensions 64 by 128.

```
(*cold-boot)                       ;8k physical processors
8192
(64 128)
```

The second example defines a configuration with a VP ratio of 2, i.e., twice as many virtual processors as physical processors.

```
(*cold-boot
   :initial-dimensions '(128 128))  ;16k virtual processors
8192
(128 128)
```

Notice that the user does not specify the VP ratio explicitly. As long as the dimensions specified are equal to either the number of physical processors attached, or to a power-of-two multiple of the number of attached processors, the proper VP ratio will be determined automatically and transparently.

NOTES

Style Note:

A typical *Lisp program has the format

```
(defun top-level ()
   (initialize-non-cm-variables)
   (*cold-boot :initial-dimensions *my-own-dimensions*)
   (initialize-cm-variables)
   (main-function))
```

There are many reasonable exceptions to this general pattern. For instance, it is possible to *define* VP sets and permanent pvars before calling ***cold–boot**. However, VP sets defined in this way remain uninstantiated and pvars likewise do not actually contain data until ***cold–boot** has been called.

Language Notes:

The *Lisp simulator permits an **:initial–dimensions** argument containing non-power-of-two dimensions, but issues a warning that such code cannot be executed on the CM-2 hardware.

If the function **initialize–character** is used to define the code, bits, or font field sizes of character pvars, it must be called immediately prior to calling ***cold–boot**, because the *Lisp global variables set by **initialize–character** are used in initializing *Lisp and the Connection Machine system. See Chapter 2, "*Lisp Global Variables" for a list of global variables controlling character attributes. See also the dictionary entry for **initialize–character.**

Usage Note:

The **:safety** keyword argument to ***cold–boot** also determines the safety level for Paris operations. If the value supplied for **:safety** is 0, Paris safety is turned off. Any other value for the **:safety** argument turns Paris safety on.

See Also:

See also the related Connection Machine initialization operator ***warm–boot.**

See also the initialization-list functions **add–initialization** and **delete–initialization.**

See also the character attribute initialization operator **initialize–character.**

compare!! *[Function]*

Performs a parallel magnitude comparison on the supplied pvars.

SYNTAX

compare!! *numeric–pvar1 numeric–pvar2*

ARGUMENTS

numeric–pvar1, numeric–pvar2
Non-complex numeric pvars to be compared.

RETURNED VALUE

compare–pvar Temporary integer pvar. In each active processor, contains either 1, 0, or –1 depending on whether the value of *numeric–pvar1* is greater than, equal to, or less than the value of *numeric–pvar2.*

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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.

EXAMPLES

```
(compare!! pvar1 pvar2)  <=>  (signum!! (-!! pvar1 pvar2))
```

---

complex!! [*Function*]

Creates and returns a complex numeric pvar.

SYNTAX

complex!! *realpart–pvar* **&optional** *imagpart–pvar*

ARGUMENTS

realpart–pvar	Non-complex numeric pvar. Real part of new complex pvar.
imagpart–pvar	Non-complex numeric pvar. Imaginary part of new complex pvar.

RETURNED VALUE

complex–pvar	Temporary complex pvar. In each active processor, contains a complex value with real and imaginary components equal to the corresponding values of *realpart–pvar* and *imagpart–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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.

Note: Because in *Lisp complex number pvars always have floating-point real and imaginary components, if the *realpart–pvar* and *imagpart–pvar* arguments are not floating-point pvars, their values are coerced to floating-point values.

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

```
(complex!! (!! 2) (!! 3))  <=>  (!! #c(2 3))

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

REFERENCES

See also these related complex pvar operators:

abs!! **cis!!**
conjugate!! **imagpart!!** **phase!!**
realpart!!

complexp!! [*Function*]

Performs a parallel test for complex values on the supplied pvar.

SYNTAX

complexp!! *pvar*

ARGUMENTS

pvar	Pvar expression. Pvar to be tested for complex values.

RETURNED VALUE

complexp–pvar	Temporary boolean pvar. Contains the value **t** in each active processor where *pvar* contains a complex value. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This predicate returns **t** in each processor whose value of *pvar* is a complex number; it returns **nil** elsewhere.

EXAMPLES

```
(complexp!! (!! #c(2 3)))  <=>  t!!
```

REFERENCES

See also these related pvar data type predicates:

booleanp!!	**characterp!!**	
floatp!!	**front-end-p!!**	**integerp!!**
numberp!!	**string-char-p!!**	**structurep!!**
typep!!		

*cond [*Macro*]
cond!! [*Function*]

Evaluate *Lisp forms with the currently selected set bound according to the results of a series of boolean tests.

SYNTAX

***cond/cond!!** (*test-pvar-1 body-forms-1*)
(*test-pvar-2 body-forms-2*)
...
(*test-pvar-n body-forms-n*)

ARGUMENTS

test-pvar-n Boolean pvar expression. Selects processors that perform the corresponding *body-forms*.

body-forms-n *Lisp forms. Evaluated with the currently selected set bound to those processors in which *test-pvar-n* has the value **t** and all previous *test-pvar* expressions have the value **nil**.

RETURNED VALUE

For ***cond**:

nil Evaluated for side effect only.

For **cond!!**:

cond-value-pvar Temporary pvar. In each active processor, contains the value returned by *value-forms-n* if and only if *test-pvar-n* has the value **t** and all previous *test-pvar* expressions have the value **nil**.

SIDE EFFECTS

For ***cond**:

None other than those of the *body-forms*.

For **cond!!**:

The returned pvar is allocated on the stack.

DESCRIPTION

The ***cond** and **cond!!** macros are parallel equivalents of the Common Lisp **cond** operation. The two operators each select groups of processors to execute different portions of *Lisp code. Unlike **cond**, however, ***cond** and **cond!!** evaluate all clauses.

The currently selected set with which each of the clauses is evaluated is determined by the *test-pvar* expressions. The forms in *body-forms-n* are evaluated with the currently selected set bound to those processors in which *test-pvar-n* has the value **t** and all previous *test-pvar* expressions have the value **nil**. Providing **t!!** as the final *test-pvar* expression selects all remaining processors.

The main difference between the ***cond** and **cond!!** is that ***cond** is used only for the side-effects of its body forms, while **cond!!** also constructs and returns a value-pvar that contains the value returned by its *body-forms*.

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

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

However, if there are no *value-forms* in a given clause, the *test-pvar* itself is used as the value of the clause, analogous to the Common Lisp **cond**.

If any active processor is not assigned a value by one of the clauses, the value of the returned pvar in that processor is **nil**, as if an implicit final clause of **(t!! nil!!)** were evaluated. An explicit final clause of the form

```
(t!! (!! default-value))
```

can be used to specify some other "default" processor value.

EXAMPLES

When the ***cond** expression

```
(*defvar result)
(*let ((mod4 (mod!! (self-address!!) (!! 4))))
   (*cond
       ((=!! mod4 (!! 0)) (*set result (!! 0)))
       ((<=!! (!! 1) mod4 (!! 2))
          (*set result (self-address!!)))
       (t!! (*set result (!! -1)))))
```

is evaluated, **result** is bound to a pvar so that is has the values displayed by:

```
(ppp result :end 10)
0 1 2 -1 0 5 6 -1 0 9
```

Similarly, when the **cond!!** expression

```
(ppp (*let ((mod4 (mod!! (self-address!!) (!! 4))))
         (cond!!   ((=!! mod4 (!! 0)) (!! 0))
                ((<=!! (!! 1) mod4 (!! 2)) (self-address!!))
                (t!! (!! -1))))
     :end 8)
```

is evaluated, it displays the values

```
0 1 2 -1 0 5 6 -1
```

NOTES

Usage Note:

Forms such as **throw, return, return-from,** and **go** may be used to exit a block or looping construct from within a processor selection operator like ***cond** or **cond!!**. However, doing so will leave the currently selected set in the state it was in at the time the non-local exit form is executed. To avoid this, use the *Lisp macro **with-css-saved**. See the dictionary entry for **with-css-saved** for more information.

Performance Note:

Currently, ***cond** and **cond!!** clauses execute serially, in the order in which they are supplied. At any given time, therefore, the number of processors active within a ***cond** clause is a subset of the currently selected set at the time the ***cond** form was entered. Providing a large number of clauses to ***cond** (and likewise **cond!!**) therefore results in potentially low overall use of processors.

Language Note:

Even if there are no selected processors, all body forms are evaluated. For example, in the expression

```
(*cond
  ((minusp!! (self-address!!)) (do-negative-actions))
  ((plusp!!  (self-address!!)) (do-positive-actions))
  ((zerop!!  (self-address!!)) (do-zero-actions))
  (t!! (when (*or t!!)
         (error "This clause cannot be executed"))))
```

the call to **do–negative–actions** is evaluated, even though no processors have a negative self address. The **do–positive–actions** call is evaluated with the currently selected set bound to all processors with a positive send address, and the **do–zero–actions** is evaluated by the single remaining processor with a send address of 0. The final **t!!** clause *is also* evaluated, even though all processors have been selected by the two preceding clauses.

Note the use, in the final **t!!** clause, of the standard *Lisp idiom **(*or t!!)** to determine whether any processors remain active. Since all processors have been selected by preceding clauses, **(*or t!!)** returns **nil**, preventing the call to **error** from being evaluated. Using an enclosing **(when (*or t!!) ...)** of this kind is a simple method of preventing evaluation of any ***cond** clause that should not be evaluated when no processors are selected.

Compiler Note:

Because an implicit **(t!! nil!!)** clause is evaluated to obtain a value for any active processor not assigned a value by one of the supplied clauses, the *Lisp compiler can occaisionally fail to compile an apparently correct **cond!!** expression, if the clauses return other than pvars of type **boolean**.

For example, given the following declarations

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

the function

```
(defun does-not-compile ()
  ;; Note that no final t!! clause is included, so an
  ;; implicit (t!! nil!!) clause is provided.
  (*set (the single-float-pvar x)
   (cond!! ((minusp!! (the single-float-pvar y)) (!! -1.0))
           ((plusp!! (the single-float-pvar y)) (!! 1.0)))))
```

does not compile. The *Lisp compiler signals an error because the implicit **(t!! nil!!)** clause returns boolean values that cannot be stored in a pvar of type **single-float-pvar**. Adding an explicit final clause that returns single-float values, as in

```
(defun does-compile ()
  ;; A final t!! clause that returns a single-float
  ;; result is included, so this function will be compiled.
  (*set (the single-float-pvar x)
        (cond!!
          ((minusp!! (the single-float-pvar y)) (!! -1.0))
          ((plusp!! (the single-float-pvar y)) (!! 1.0))
           (t!! (!! 0.0))
          )))
```

allows this function to compile.

REFERENCES

See also the related operators

***all**	***case**	**case!!**	***ecase**	**ecase!!**	***if**	**if!!**
***unless**	***when**	**with-css-saved**				

conjugate!! [Function]

Calculates in parallel the complex conjugate of the supplied pvar.

SYNTAX

conjugate!! *numeric–pvar*

ARGUMENTS

numeric–pvar Numeric pvar. Pvar for which the complex conjugate is calculated.

RETURNED VALUE

conjugate–pvar Temporary numeric pvar. Contains in each active processor the complex conjugate of the corresponding value of *numeric–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

Returns a temporary pvar whose value in each processor is the complex conjugate of the corresponding value of *numeric–pvar.* (The conjugate of a complex number is another complex number with the same real component and the negation of the imaginary component of the original number.)

```
(conjugate!! (!! #c(4 5)))  <=>  (!! #c(4 -5))
```

REFERENCES

See also these related complex pvar operators:

abs!!	**cis!!**	**complex!!**
imagpart!!	**phase!!**	**realpart!!**

---

copy-seq!! [*Function*]

Returns a copy of the supplied sequence pvar.

SYNTAX

copy-seq!! *sequence-pvar*

ARGUMENTS

sequence-pvar Sequence pvar. Pvar to be copied. Must be a vector pvar.

RETURNED VALUE

copy-seq-pvar Temporary sequence pvar. Contains in each active processor a copy of the corresponding value of *sequence-pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function returns a copy of *sequence-pvar*. For example,

```
(copy-seq!! data-pvar)
```

returns a copy of **data-pvar** as a temporary pvar on the stack.

EXAMPLES

```
(*defvar seq-pvar (!! #(1 2 3 4)))

(ppp seq-pvar :end 5)
#(1 2 3 4) #(1 2 3 4) #(1 2 3 4) #(1 2 3 4) #(1 2 3 4)
```

```
(*let ((seq-copy (copy-seq!! seq-pvar)))
   (*setf (pref seq-copy 2) #(4 3 2 1))
   (ppp seq-copy :end 5))
#(1 2 3 4) #(1 2 3 4) #(4 3 2 1) #(1 2 3 4) #(1 2 3 4)

(ppp seq-pvar :end 5)
#(1 2 3 4) #(1 2 3 4) #(1 2 3 4) #(1 2 3 4) #(1 2 3 4)
```

NOTES

Compiler Note:

The *Lisp compiler does not compile this operation.

REFERENCES

See also these related *Lisp sequence operators:

***fill**	**length!!**	
***nreverse**	**reduce!!**	**reverse!!**
subseq!!		

See also the generalized array mapping functions **amap!!** and ***map**.

cos!!, cosh!! [*Function*]

Take the cosine and hyperbolic cosine of the supplied pvar.

SYNTAX

cos!! *radians–pvar*
cosh!! *radians–pvar*

ARGUMENTS

radians–pvar	Numeric pvar. Angle, in radians, for which the cosine (hyperbolic cosine) is calculated.

RETURNED VALUE

result–pvar	Temporary numeric pvar. In each active processor, contains the cosine (hyperbolic cosine) of *radians–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The function **cos!!** returns the cosine of *radians–pvar.*
The function **cosh!!** returns the hyperbolic cosine of *radians–pvar.*

EXAMPLES

```
(cos!! (!! 0))      <=>  (!! 1)
(cosh!! (!! 1))     <=>  (!! 1.5430806)
```

---

count!!, count–if!!, count–if–not!! [*Function*]

Perform a parallel count on a sequence pvar, returning in each processor the number of sequence elements that match a given item or pass/fail a test.

SYNTAX

count!!	*item–pvar sequence–pvar*	
	&key :test :test–not	**:start :end :key :from–end**
count–if!!	*test sequence–pvar* **&key**	**:start :end :key :from–end**
count–if–not!!	*test sequence–pvar* **&key**	**:start :end :key :from–end**

ARGUMENTS

item–pvar	Pvar expression. Item to match in *sequence–pvar*. Must be of the same type as the elements of *sequence–pvar*.
test	One-argument pvar test. Used to test elements of *sequence–pvar*.
sequence–pvar	Sequence pvar. Contains sequences to be searched.
:test	Two-argument pvar predicate. Test used in comparisons. Indicates a match by returning a non-**nil** result. Defaults to **eql!!**.
:test–not	Two-argument pvar predicate. Test used in comparisons. Indicates a match by returning a **nil** result.
:start	Integer pvar. Zero-based index of sequence element at which counting starts in each processor. If not specified, counting begins with first element.
:end	Integer pvar. Zero-based index of sequence element at which counting ends in each processor. If not specified, counting continues to end of sequence.
:key	One-argument pvar accessor function. Applied to *sequence–pvar* before counting is performed.
:from–end	Boolean. Whether to begin search from end of sequence in each processor. **Note:** This argument is currently ignored.

RETURNED VALUE

count-pvar Temporary integer pvar. In each active processor, contains the number of matching elements of *sequence-pvar*. If no matching elements are found, **(!! 0)** is returned

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

These functions are the parallel equivalent of the Common Lisp **count**, **count-if**, and **count-if-not** functions.

In each processor, the function **count!!** searches *sequence-pvar* for elements that match *item-pvar*. It returns a pvar containing a count of the matching elements found in each processor. Elements of *sequence-pvar* are tested against *item-pvar* with the **eql!!** operator unless another comparison operator is supplied as either of the **:test** or **:test-not** keyword arguments. The keywords **:test** and **:test-not** may not be used together. A lambda form that takes two pvar arguments and returns a boolean pvar result may be supplied as either the **:test** and **:test-not** argument.

In each processor, the function **count-if!!** searches *sequence-pvar* for elements that satisfy the supplied *test*. It returns a pvar containing a count of the sequence elements found in each processor. A lambda form that takes a single pvar argument and returns a boolean pvar result may be supplied as the *test* argument.

In each processor, the function **count-if-not!!** searches *sequence-pvar* for elements that fail the supplied *test*. It returns a pvar containing a count of the sequence elements found in each processor. A lambda form that takes a single pvar argument and returns a boolean pvar result may be supplied as the *test* 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.

Arguments to the keywords **:start** and **:end** define a subsequence to be operated on in each processor.

The **:key** keyword accepts a user-defined function used to extract a search key from *sequence-pvar*. This key function must take one argument: an element of *sequence-pvar*.

NOTES

Compiler Note:

The *Lisp compiler does not compile this operation.

REFERENCES

The functions **count!!**, **count-if!!**, and **count-if-not!!** are similar to the *Lisp functions **find!!**, **find-if!!**, and **find-if-not!!**. Unlike the **find!!** functions, however, a **count!!** search continues until *sequence-pvar* is exhausted.-

These functions are members of a group of similar sequence operators, listed below:

count!!	**count-if!!**	**count-if-not!!**
find!!	**find-if!!**	**find-if-not!!**
nsubstitute!!	**nsubstitute-if!!**	**nsubstitute-if-not!!**
position!!	**position-if!!**	**position-if-not!!**
substitute!!	**substitute-if!!**	**substitute-if-not!!**

See also the generalized array mapping functions **amap!!** and ***map**.

create–geometry [*Function*]

Creates and returns a geometry object.

SYNTAX

create–geometry &key :dimensions :weights :ordering :on–chip–bits :off–chip–bits

ARGUMENTS

:dimensions Required argument. A list of integers, each of which must be a power of 2. Defines the size of each dimension specified by the returned geometry object.

:weights List of integers, one for each dimension. Indicates relative frequency of NEWS communication expected for each dimension. Default value assigns equal weight to each dimension. If a **:weights** argument is supplied, neither of the **:on–chip–bits** and **:off–chip–bits** arguments should be supplied.

:ordering List of symbols, one for each dimension. Only the symbols **:news–order** and **:send–order** may be supplied in the list. Controls optimization of address translation for each dimension. Default value assigns the symbol **:news–order** to each dimension.

:on–chip–bits, :off–chip–bits
Lists of integers, one for each dimension. Determine processor address translation. These arguments are provided in *Lisp as a direct hook into Paris.

RETURNED VALUE

geometry–obj Geometry object, suitable as an argument to ***cold–boot**, **def–vp–set**, **create–vp–set**, **set–vp–set–geometry**, and **allocate–processors–for–vp–set**.

SIDE EFFECTS

None.

DESCRIPTION

The **create–geometry** function creates and returns a data structure known as a geometry object. Geometry objects are used to define the shape of virtual processor sets. In addition, they permit control over interprocessor communication speed within a VP set. This can be particularly useful when it is critical to optimize the performance of scanning operations along specific dimensions of a VP set.

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 specify an *n*-dimensional hypercube of virtual processors. The product of the dimensions must be a power of two multiple of the physical machine size.

If supplied, the value of **:weights** specifies the relative frequency of NEWS communication along each dimension. Given the specified weighting, the Connection Machine allocates virtual processors for optimal performance.

For example, consider a three-dimensional VP set in which near neighbor communication is estimated to be twice as frequent in dimension 1 as in either dimension 0 or 2. In this case, the **:weights** argument should be the list **'(1 2 1)**.

If supplied, the value of **:ordering** controls optimization of address translation for each dimension. For dimensions specified as **:news–order**, send addresses are gray-coded and mapped into NEWS addresses. This ensures that processors with neighboring send addresses are actually NEWS neighbors within the machine. For dimensions specified as **:send–order**, no special address translation is done. Processors with neighboring geometry positions along these dimensions have neighboring send addresses.

The **:on–chip–bits** and **:off–chip–bits** arguments together specify a pair of bitmasks that map send addresses into NEWS addresses, providing maximum control over interprocessor communication patterns at the hardware level. These arguments are provided in *Lisp as a direct hook into Paris.

EXAMPLES

The **create–geometry** function is most often used to specify the geometry of a VP set. For example,

```
(def-vp-set three-dee nil
   :geometry-definition-form
      (create-geometry :dimensions '(64 128 8)
         :weights '(1 3 1)
         :ordering '(:send-order :news-order :send-order)))
```

defines a three-dimensional VP set, **three–dee**. The geometry object returned by **create–geometry** specifies that NEWS communication will take place along dimension 1 of **three–dee** three times as often as along either dimension 0 or 2. Also, the geometry object specifies that only dimension 1 of **three–dee** should be optimized for NEWS addressing.

The **create–geometry** function may also be used to instantiate an existing flexible VP set, as in

```
(def-vp-set flexible-vp-set nil
   :geometry-definition-form nil)

(allocate-processors-for-vp-set
   flexible-vp-set
   nil
   :geometry (create-geometry :dimensions '(32 128 64)))
```

which assigns a three-dimensional geometry to the VP set **flexible–vp–set**.

Finally, the **create–geometry** function may be used to specify the geometry of the ***default–vp–set***. For example,

```
(*cold-boot :initial-geometry-definition
   (create-geometry :dimensions '(32 128)))
```

defines a two-dimensional default VP set.

NOTES

The **create–geometry** function makes it possible to optimize a VP set geometry for NEWS communication along certain dimensions and for general send-address communication along other dimensions.

The **:weights**, **:ordering**, **:on–chip–bits**, and **:off–chip–bits** arguments default to reasonable values if not specified. These arguments affect only the run-time performance of interprocessor communication. They do not affect the data transmitted in any way.

The majority of *Lisp users will never need to use the **:on–chip–bits** and **:off–chip–bits** arguments; the **:weights** argument is usually sufficient.

REFERENCES

See the definitions of ***cold–boot**, **def–vp–set**, **create–vp–set**, **let–vp–set**, **set–vp–set–geometry**, and **allocate–processors–for–vp–set** for discussions on how to use geometry objects.

See the Concepts section of the *Paris Reference Manual* for more information on the effect of address orderings. Also in the *Paris Reference Manual*, see the dictionary entry for **CM:create–detailed–geometry.**

create–segment–set!! *[Function]*

Creates and returns a segment set structure pvar that defines a segment set.

SYNTAX

create–segment–set!! &key :start–bit :end–bit

ARGUMENTS

:start–bit	Boolean pvar. Specifies processors that start a segment. If not supplied, starting processors are determined from **:end–bit** argument.
:end–bit	Boolean pvar. Specifies processors that end a segment. If not supplied, starting processors are determined from **:start–bit** argument.

RETURNED VALUE

segment–set–obj	Segment set pvar, suitable for use as the third argument in a call to the **segment–set–scan!!** operation.

SIDE EFFECTS

None.

DESCRIPTION

This function returns a segment set pvar suitable for use as the third argument in a call to the **segment–set–scan!!** operation.

The two keyword 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–bit** argument may be a pvar containing the value **t** in each processor that starts a segment and **nil** in all other processors. Alternatively, to signify that the **:end–bit** argument is to be used to determine where the segments start, **:start–bit** may be **nil!!** or simply not supplied.

Likewise, the **:end–bit** argument may be a pvar containing the value **t** in each processor that ends a segment and **nil** in all other processors. To signify that the **:start–bit** argument is to be used to determine where the segments end, **:end–bit** may be **nil!!** or simply not supplied.

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 not supplied, completely adjacent segments are defined.

When constructing pvars to supply as **:start–bit** or **:end–bit** arguments, 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 contain the **:start–bit** and **:end–bit** argument pvars supplied 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.

REFERENCES

See also these related segment set operators:

segment–set–scan!!

segment–set–end–bits	**segment–set–end–bits!!**
segment–set–end–address	**segment–set–end–address!!**
segment–set–start–bits	**segment–set–start–bits!!**
segment–set–start–address	**segment–set–start–address!!**

segment–set–processor–not–in–any–segment

segment–set–processor–not–in–any–segment!!

create–vp–set *[Function]*

Creates and returns a VP set definition object.

SYNTAX

create–vp–set *dimensions* **&key :geometry**

ARGUMENTS

dimensions	Either **nil** or a list of integers, each of which is a power of 2. Specifies the dimension sizes of the VP set object returned.
:geometry	Either **nil** or a geometry object as returned by **create–geometry**. Specifies geometry of VP set object returned.

RETURNED VALUE

vp–set–obj	VP set object. Descriptor object for newly created VP set.

SIDE EFFECTS

None.

DESCRIPTION

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 integral 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 the **:geometry** argument is provided, it incorporates information about the dimensions of the VP set being defined. (See the definition of **create–geometry** for more details.)

EXAMPLES

The *Lisp forms

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

create two VP sets. The first, **x**, is created with a 3-dimensional configuration. The second, **y**, is created with a 5-dimensional configuration, using the function **vp–set–dimensions** to obtain the dimension sizes specified for the **x** VP set.

The **create–vp–set** function is normally used during program execution, not at top level. Below is an example of how **create–vp–set** might be used in a program.

```
(defun make-2d-vp-set (linear-vp-set n linear-pvar)
  (let ((new-vp-set (create-vp-set (list n n))))
    (*with-vp-set new-vp-set
      (*let ((new-pvar (!! 0)))
        (*with-vp-set linear-vp-set
          (*when (<!! (self-address!!) (!! n))
            (*pset :no-collisions linear-pvar new-pvar
              (cube-from-vp-grid-address!!
               new-vp-set (self-address!!) (self-address!!)))
            (*with-vp-set new-vp-set
              (ppp new-pvar :mode :grid :end '(4 4)))))))
    (deallocate-vp-set new-vp-set)))
```

This example uses **create–vp–set** to create an **n** x **n** vp set, **new–vp–set**. It then creates a pvar, **new–pvar**, within the two-dimensional **new–vp–set**, and uses ***pset** to store the first **n** elements of **linear–pvar** into the main diagonal elements of **new–pvar**. With **new–vp–set** selected, a function is called to perform an operation on the **new–pvar**, and finally **deallocate–vp–set** is called to deallocate the **new–vp–set**.

Because **n** is used to determine the dimensions of VP sets, **n** must be a power of two.

An example of how this function might be called is:

```
(defparameter vp-set-size 32)
(def-vp-set 1d-vp-set (list vp-set-size))
  :*defvars ((1d-pvar (self-address!!))))

(make-2d-vp-set 1d-vp-set vp-set-size 1d-pvar)

0 0 0 0
0 1 0 0
0 0 2 0
0 0 0 3
```

REFERENCES

See also the following VP set definition and deallocation operators:

def–vp–set **let–vp–set**
deallocate–def–vp–sets **deallocate–vp–set**

See also the following geometry definition operator:

create–geometry

The following math utilities are useful in defining the size of VP sets:

next–power–of–two–>= **power–of–two–p**

See also the following flexible VP set operators:

allocate–vp–set–processors **allocate–processors–for–vp–set**
deallocate–vp–set–processors **deallocate–processors–for–vp–set**
set–vp–set–geometry **with–processors–allocated–for–vp–set**

These operations are used to select the current VP set:

set–vp–set ***with–vp–set**

See also the following VP set information operations:

dimension–size **dimension–address–length**
describe–vp–set **vp–set–deallocated–p**
vp–set–dimensions **vp–set–rank**
vp–set–total–size **vp–set–vp–ratio**

cross–product [*Function*]

Returns the cross product of two front-end vectors.

SYNTAX

cross–product *vector1 vector2*

ARGUMENTS

vector1, vector2 Front-end vectors, for which the cross product is returned.

RETURNED VALUE

cross–prod–vector Front-end vector. Cross product of *vector1* and *vector2*.

SIDE EFFECTS

None.

DESCRIPTION

This is the serial (front end) equivalent of **cross–product!!**. The cross product of the two vectors is computed. The result is returned as a vector. The vector arguments must be of length 3

```
(cross-product #(1 2 3) #(4 5 6)) => #(-3 6 -3)
```

REFERENCES

This function is one of a number of front-end vector operators, listed below:

cross–product	**dot–product**	**v+ v– v* v/**	
v+–constant	**v—constant**	**v*–constant**	**v/–constant**
vabs	**vabs–squared**	**vceiling**	**vector–normal**
vfloor	**vround**	**vscale**	
vscale–to–unit–vector		**vtruncate**	

These functions are the serial equivalents of the corresponding vector pvar operations. See Chapter 1, "*Lisp Overview," of this Dictionary for a list of these functions.

cross–product!! [*Function*]

Performs a parallel cross product operation on the supplied vector pvars.

SYNTAX

cross–product!! *vector–pvar1 vector–pvar2*

ARGUMENTS

vector–pvar1, vector–pvar2
: Vector pvars, for which the cross product is returned.

RETURNED VALUE

cross–prod–vector–pvar
: Temporary vector pvar. In each active processor, contains the cross product of the corresponding values of *vector–pvar1* and *vector–pvar2*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

In each processor, the cross product of the two vector pvars is computed. The result is returned as a vector pvar.

```
(cross-product!! (!! #(1 2 3)) (!! #(4 5 6)))  <=>
(!! #(-3 6 -3))
```

The arguments *vector–pvar1* and *vector–pvar2* must be pvar vectors of length 3.

NOTES

Compiler Note:

The *Lisp compiler does not compile this operation.

REFERENCES

This function is one of a number of related vector pvar operators, listed below:

cross–product!!	**dot–product!!**	**v+!! v–!! v*!! v/!!**	
v+scalar!!	**v–scalar!!**	**v*scalar!!**	**v/scalar!!**
vabs!!	**vabs–squared!!**	**vector–normal!!**	**vscale!!**
vscale–to–unit–vector!!		***vset–components**	

cube–from–grid–address [*Function*]

Converts a grid (NEWS) address in the current VP set into a send (cube) address.

SYNTAX

cube–from–grid–address *coordinate* **&rest** *coordinates*

ARGUMENTS

coordinate, coordinates
A set of integers representing a grid (NEWS) address in the current VP set. The number of *coordinates* supplied must equal the rank of the current VP set.

RETURNED VALUE

send–address Integer. The send (cube) address corresponding to the set of *coordinates*.

SIDE EFFECTS

None.

DESCRIPTION

This function translates a series of integers specifying the grid (NEWS) address of a single processor in the current VP set into a single integer specifying the send (cube) 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.

EXAMPLES

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) has a send (cube) address of 1036.

NOTES

Note that the send (cube) address corresponding to a particular grid address is not predictable from the grid address values alone. It also depends on the geometry of the current VP set, on the number of physical processors attached, and on the system software version in use. In particular, the relationship between send and grid addresses in the *Lisp simulator is different from that of the actual CM-2 hardware.

It is an error to rely on a specific, fixed relation between send and grid addresses except as provided by *Lisp address conversion functions such as **cube–from–grid–address**, **cube–from–vp–grid–address**, **grid–from–cube–address**, and **grid–from–vp–cube–address.**

REFERENCES

See also these related send and grid address translation operators:

cube–from–grid–address!!
cube–from–vp–grid–address **cube–from–vp–grid–address!!**
grid–from–cube–address **grid–from–cube–address!!**
grid–from–vp–cube–address **grid–from–vp–cube–address!!**
self–address!! **self–address–grid!!**

cube–from–grid–address!! [*Function*]

Performs a parallel conversion from grid (NEWS) addresses in the current VP set to send (cube) addresses .

SYNTAX

cube–from–grid–address!! *coordinate–pvar* **&rest** *coordinate–pvars*

ARGUMENTS

coordinate–pvar, coordinate–pvars
A series of integer pvars representing, in each processor, a grid (NEWS) address in the current VP set. The number of *coordinate–pvars* supplied must equal the rank of the current VP set.

RETURNED VALUE

send–address–pvar Temporary integer pvar. In each active processor, contains the send (cube) address corresponding to the values of the *coordinate–pvars*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function translates a series of *coordinate–pvars*, specifying a grid (NEWS) address in each processor in the current VP set, into a single pvar that contains the corresponding send (cube) address in each processor.

This is the parallel equivalent of **cube–from–grid–address**.

EXAMPLES

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

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

Here, the send (cube) address of the processor located at coordinates (10, 20, 30), 1036, is returned in all active processors.

NOTES

Note that the send (cube) address corresponding to a particular grid (NEWS) address is not predictable from the grid (NEWS) address values alone. It also depends on the geometry of the current VP set, on the number of physical processors attached, and on the system software version in use.

For example, on the CM hardware, the expression

```
(*cold-boot :initial-dimensions '(32 16))
(ppp (cube-from-grid-address!!
         (self-address-grid!! (!! 0))
         (self-address-grid!! (!! 1)))
       :mode :grid :end '(4 4))
```

may display the following:

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

On the *Lisp simulator, the same code displays

```
0 16 32 48
1 17 33 49
2 18 34 50
3 19 35 51
```

It is an error to rely on a specific, fixed relation between send and grid addresses except as provided by *Lisp address conversion functions such as **cube-from-grid-address!!**, **cube-from-vp-grid-address!!**, **grid-from-cube-address!!**, and **grid-from-vp-cube-address!!**.

REFERENCES

See also these related send and grid address translation operators:

cube–from–grid–address

cube–from–vp–grid–address	**cube–from–vp–grid–address!!**
grid–from–cube–address	**grid–from–cube–address!!**
grid–from–vp–cube–address	**grid–from–vp–cube–address!!**
self–address!!	**self–address–grid!!**

cube–from–vp–grid–address [*Function*]

Converts a grid (NEWS) address in the specified VP set into a send (cube) address.

SYNTAX

cube–from–vp–grid–address *vp–set coordinate* **&rest** *coordinates*

ARGUMENTS

vp–set VP set object. VP set for which the supplied *coordinates* are converted. Must be both defined and instantiated.

coordinate, coordinates
A set of integers representing a grid (NEWS) address in *vp–set.* The number of *coordinates* supplied must equal the rank of *vp–set.*

RETURNED VALUE

send–address Integer. The send (cube) address corresponding to the set of *coordinates*.

SIDE EFFECTS

None.

DESCRIPTION

This function translates a series of integer *coordinates* that specify the grid (NEWS) address of a single processor in *vp–set* into a single integer specifying the send (cube) address of that processor.

EXAMPLES

For example, assuming the VP set **my–vp** has a three-dimensional geometry,

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

Here, the processor located at coordinates (10, 20, 30) in the **my–vp** VP set has a send (cube) address of 1036. This means that the processor at coordinates (10, 20,30) in **my–vp** can be accessed directly via the send address 1036, as in

```
(pref (self-address!!) 1036)  =>  1036
```

Using this conversion mechanism, it is unnecessary to make **my–vp** the current VP set in order to access processors via grid addresses within **my–vp**, as in

```
(*with-vp-set my-vp
   (pref (self-address!!) (grid 10 20 30)))  =>  1036
```

NOTES

Note that the send (cube) address corresponding to a particular grid (NEWS) address is not predictable from the grid (NEWS) address values alone. It also depends on the geometry of the current VP set, on the number of physical processors attached, and on the system software version in use.

It is an error to rely on a specific, fixed relation between send and grid addresses except as provided by *Lisp address conversion functions such as **cube–from–grid–address**, **cube–from–vp–grid–address**, **grid–from–cube–address**, and **grid–from–vp–cube–address**.

REFERENCES

See also these related send and grid address translation operators:

cube–from–grid–address	**cube–from–grid–address!!**
cube–from–vp–grid–address!!	
grid–from–cube–address	**grid–from–cube–address!!**
grid–from–vp–cube–address	**grid–from–vp–cube–address!!**
self–address!!	**self–address–grid!!**

cube–from–vp–grid–address!! [*Function*]

Performs a parallel conversion from grid (NEWS) addresses in the specified VP set into send (cube) addresses.

SYNTAX

cube–from–vp–grid–address!! *vp–set coordinate–pvar* **&rest** *coordinate–pvars*

ARGUMENTS

vp–set — VP set object. VP set for which the coordinates in the supplied *coordinate–pvars* are converted. Must be both defined and instantiated.

coordinate–pvar, coordinate–pvars — A set of integer pvars representing in each processor a grid (NEWS) address in *vp–set*. The number of *coordinate–pvars* supplied must equal the rank of *vp–set*.

RETURNED VALUE

send–address–pvar — Temporary integer pvar. In each active processor, contains the send (cube) address corresponding to the values of the *coordinate–pvars*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function converts a series of *coordinate–pvars*, specifying the grid (NEWS) addresses of processors in *vp–set*, into a single pvar that specifies the send (cube) addresses of those processors. This is the parallel equivalent of **cube–from–vp–grid–address**.

EXAMPLES

For example, assuming the VP set **my-vp** has a three-dimensional geometry,

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

Here, the send (cube) address of the processor located at coordinates (10, 20, 30) in the **my-vp** VP set, 1036, is returned in all active processors.

NOTES

Note that the send (cube) address corresponding to a particular grid (NEWS) address is not predictable from the grid (NEWS) address values alone. It also depends on the geometry of the current VP set, on the number of physical processors attached, and on the system software version in use.

For example, on the CM hardware, the expression

```
(def-vp-set two-dim '(32 16))
(ppp (cube-from-vp-grid-address!! two-dim
       (self-address-grid!! (!! 0))
       (self-address-grid!! (!! 1)))
     :mode :grid :end '(4 4))
```

may display the following:

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

On the *Lisp simulator, the same code displays

0 16 32 48
1 17 33 49
2 18 34 50
3 19 35 51

It is an error to rely on a specific, fixed relation between send and grid addresses except as provided by *Lisp address conversion functions such as **cube-from-grid-address!!**, **cube-from-vp-grid-address!!**, **grid-from-cube-address!!**, and **grid-from-vp-cube-address!!**.

REFERENCES

See also these related send and grid address translation operators:

cube–from–grid–address	**cube–from–grid–address!!**
cube–from–vp–grid–address	
grid–from–cube–address	**grid–from–cube–address!!**
grid–from–vp–cube–address	**grid–from–vp–cube–address!!**
self–address!!	**self–address–grid!!**

*deallocate [*Function*]

Deallocates a global pvar.

SYNTAX

***deallocate** *pvar*

ARGUMENTS

pvar	Pvar expression. The global pvar to deallocate. Must have been allocated by **allocate!!**.

RETURNED VALUE

nil	Evaluated for side effect.

SIDE EFFECTS

Deallocates the global pvar *pvar*, freeing the heap memory assigned to it on the CM.

DESCRIPTION

This function deallocates the supplied global *pvar*, which must have been allocated by **allocate!!**.

EXAMPLES

```
(allocate!! global-pvar)
...
;code using global-pvar
;
(*deallocate global-pvar)
```

NOTES

It is an error to use a pvar after it has been deallocated. The order in which pvars are deallocated does not matter.

Global pvars and permanent pvars are allocated on the CM heap. In contrast to global pvars, which are allocated by **allocate!!** and deallocated with **deallocate***, permanent pvars, are allocated by ***defvar** and must be deallocated by the function ***deallocate–*defvars**.

REFERENCES

See also the pvar allocation and deallocation operations

allocate!!	**array!!**	
***deallocate–*defvars**	***defvar**	
front–end!!	***let**	***let***
make–array!!	**typed–vector!!**	**vector!!**
!!		

See the *Lisp glossary for definitions of the different kinds of pvars that are allocated on the CM stack and heap.

*deallocate–*defvars [*Function*]

Deallocates some or all permanent pvars allocated by ***defvar**.

SYNTAX

***deallocate–*defvars &rest** *pvar–names*

ARGUMENTS

pvar–names	A series of symbols naming permanent pvars that have been allocated by ***defvar**, or one of the symbols **:prompt**, **:all**, **:all–noconfirm**, or **nil**. Specifies the pvars to deallocate.

RETURNED VALUE

nil	Evaluated for side-effect.

SIDE EFFECTS

Deallocates the permanent pvars specified by *pvar–names*, freeing the CM heap memory they have occupied.

DESCRIPTION

This function deallocates the pvars specified in *pvar–names*.

If *pvar–names* is **nil** or **:prompt**, the user is prompted with the name of each pvar ever declared with ***defvar**, and given the option of deallocating the pvar, or of skipping it and going on to the next pvar. Skipped pvars are not deallocated.

If *pvar–names* is **:all**, then after the user is prompted for confirmation all pvars allocated with ***defvar** are deallocated.

If *pvar–names* is **:all–noconfirm**, then all pvars declared with ***defvar** are deallocated.

EXAMPLES

Here are some sample uses:

```
(*deallocate-*defvars 'foo)       ;delete foo pvar

(*deallocate-*defvars 'foo 'bar) ;delete foo and bar pvars

(*deallocate-*defvars :prompt)    ;get prompted for pvars
                                  ;to delete

(*deallocate-*defvars)            ;get prompted for pvars
                                  ;to delete

(*deallocate-*defvars :all)       ;delete all pvars declared
                                  ;with *defvar
```

NOTES

Before deallocating any permanent pvar, be certain that no library functions depend on that pvar.

The two predefined pvars, **t!!** and **nil!!**, can never be deallocated.

Global pvars and permanent pvars are allocated on the CM heap. In contrast to global pvars, which are allocated by **allocate!!** and deallocated with **deallocate***, permanent pvars, are allocated by ***defvar** and must be deallocated by the function ***deallocate–*defvars**.

REFERENCES

See also the pvar allocation and deallocation operations

allocate!!	**array!!**	
***deallocate**	***defvar**	
front–end!!	***let**	***let***
make–array!!	**typed–vector!!**	**vector!!**
!!		

See the *Lisp glossary for definitions of the different kinds of pvars that are allocated on the CM stack and heap.

deallocate–def–vp–sets *[Function]*

Deallocates some or all permanent VP sets, which were defined using **def–vp–set**.

SYNTAX

deallocate–def–vp–sets &rest *vp–sets*

ARGUMENTS

vp–sets	VP sets to be deallocated, or the keyword **:all**.

RETURNED VALUE

nil	Evaluated for side effect.

SIDE EFFECTS

Deallocates VP sets specified by *vp–sets* using **deallocate–vp–set**.

DESCRIPTION

This function deallocates each of the supplied *vp–sets*, using **deallocate–vp–set**. If the *vp–sets* argument is the single keyword **:all**, all VP sets defined using **def–vp–set** are deallocated.

EXAMPLES

```
(deallocate-def-vp-sets vp-set-1 vp-set2)
(deallocate-def-vp-sets :all)
```

REFERENCES

See the *Lisp glossary for definitions of the kinds of VP sets that may be allocated and deallocated.

See also the following VP set definition and deallocation operators:

def–vp–set	**create–vp–set**
let–vp–set	**deallocate–vp–set**

deallocate–geometry [*Function*]

Deallocates an existing geometry object.

SYNTAX

deallocate–geometry *geometry*

ARGUMENTS

geometry Geometry object. Geometry to be deallocated.

RETURNED VALUE

nil Evaluated for side effect.

SIDE EFFECTS

The geometry specified by *geometry* is deallocated.

DESCRIPTION

The geometry specified by *geometry* must be a geometry object, as created by the *Lisp operator **create–geometry**. The specified *geometry* is deallocated.

EXAMPLES

```
(setq my-geo (create-geometry :dimensions '(32 16)))
(deallocate-geometry my-geo)
```

NOTES

It is an error to delete a geometry that is currently associated with an active VP set.

deallocate–processors–for–vp–set [*Function*]
deallocate–vp–set–processors [*Function*]

Deinstantiates a flexible VP set, deallocating any associated pvars.

SYNTAX

deallocate–processors–for–vp–set *vp–set* **&key :ok–if–not–instantiated**

ARGUMENTS

vp–set Flexible VP set. Virtual processor set defined with **def–vp–set**.

:ok–if–not–instantiated
Boolean value. Determines whether error is signalled if *vp–set* does not currently have any processors allocated.

RETURNED VALUE

nil Evaluated for side effect.

SIDE EFFECTS

Deinstantiates VP set, and deallocates CM memory assigned to associated pvars. Definitions of permanent pvars are retained, and these pvars are reallocated when the VP set is reinstantiated.

DESCRIPTION

Deallocates all processors previously allocated for the specified VP set by a call to **allocate–processors–for–vp–set.**

The *vp–set* parameter must be a flexible VP set for which processors have been allocated by either **allocate–processors–for–vp–set** or **allocate–vp–set–processors.** The specified VP set itself is not destroyed and the definitions of any associated permanent pvars are retained. However, all other pvars, including global pvars created by

allocate!!, are deallocated and destroyed by a call to the **deallocate–processors–for–vp–set** function.

The **:ok–if–not–instantiated** keyword takes a boolean argument and defaults to **nil**. It determines whether or not an error is signaled if the provided VP set is not instantiated at the time of the call.

EXAMPLES

This example shows how **allocate–processors–for–vp–set**, along with its companion function **deallocate–processors–for–vp–set**, may be used to instantiate a flexible VP set several times with a different geometry at each invocation.

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

(defun process-files (&rest diskfiles)
  (*cold-boot)
  ;;; at this point, disk-data-pvar has no memory allocated
  ;;; on the CM
  (dolist (file diskfiles)
    (let ((elements (read-number-of-elements-in file)))
      (allocate-processors-for-vp-set disk-data
         (list (next-power-of-two->= elements)))
      ;;; now disk-data-pvar has CM memory allocated
      (let ((array-of-data (read-data-from-disk file)))
        (array-to-pvar array-of-data disk-data-pvar
                       :cube-address-end elements)
        (process-data-in-cm disk-data disk-data-pvar))
      (deallocate-processors-for-vp-set disk-data))))
```

NOTES

The function **deallocate–vp–set–processors** is an obsolete alias for the function **deallocate–processors–for–vp–set**, and behaves identically.

REFERENCES

See the *Lisp glossary for a definition of *flexible VP set* and for definitions of all the kinds of VP sets that may be allocated and deallocated.

See also the following flexible VP set operators:

allocate–vp–set–processors	**allocate–processors–for–vp–set**
set–vp–set–geometry	**with–processors–allocated–for–vp–set**

See also the following VP set definition and deallocation operators:

def–vp–set	**create–vp–set**	**let–vp–set**
deallocate–def–vp–sets	**deallocate–vp–set**	

deallocate–vp–set *[Function]*

Deallocates a permanent or temporary VP set and its associated pvars.

SYNTAX

deallocate–vp–set *vp–set* **&optional** *deallocate–geometry–p*

ARGUMENTS

vp–set VP set object. VP set to be deallocated.

deallocate–geometry–p
Scalar boolean value. Determines whether the geometry object associated with the VP set is deallocated.

RETURNED VALUE

returned–value Returned value.

DESCRIPTION

This function deallocates the supplied *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 as well. If *vp–set* was defined by **def–vp–set**, then the symbol that names the VP set is made unbound.

The optional argument, *deallocate–geometry–p,* is a boolean value that determines whether the geometry object associated with the specified VP set is to be deallocated. The default is **t**; the assocated geometry object is deallocated by default.

NOTES

Usage Note

The **let–vp–set** form automatically calls **deallocate–vp–set** using the default argument to *deallocate–geometry–p.* Do not assign a geometry object that should be preserved to a temporary VP set created with **let–vp–set**.

REFERENCES

See the *Lisp glossary for definitions of permanent and temporary VP sets.

See also the following VP set definition and deallocation operators:

def–vp–set	**create–vp–set**
let–vp–set	**deallocate–def–vp–sets**

*decf *[Macro]*

Destructively decrements each value of the supplied pvar.

SYNTAX

***decf** *numeric–pvar* **&optional** *value–pvar*

ARGUMENTS

numeric–pvar	Pvar expression. Pvar to be decremented.
value–pvar	Numeric pvar. Amount to subtract from *numeric–pvar*. Defaults to **(!! 1)**.

RETURNED VALUE

nil	Evaluated for side effect.

SIDE EFFECTS

Destructively decrements each value of *pvar* by the corresponding value of *value–pvar*.

DESCRIPTION

Destructively decrements each element of *numeric–pvar* by the corresponding value of *value–pvar*. The *value–pvar* argument defaults to **(!! 1)**.

EXAMPLES

```
(*decf count-pvar (!! 3))
```

NOTES

Usage Note:

A call to the ***decf** macro expands as follows:

```
(*decf data-pvar (!! 4))
    ==>
(*setf data-pvar (-!! data-pvar (!! 4)))
```

For this reason, the *numeric-pvar* must be a modifiable pvar, such as a permanent, global, or local pvar. It is an error to supply a temporary pvar as the *numeric-pvar* to ***decf**.

REFERENCES

See also the related macro ***incf**.

The function **1-!!** can be used to non-destructively perform a subtraction by 1 on its argument pvar. See the dictionary entry on **1-!!** for more information.

*defsetf [*Macro*]

Assigns an update function to be used whenever ***setf** is called on the specified access function.

SYNTAX

***defsetf** *accessor-function update-function*

ARGUMENTS

accessor-function	Symbol. The name of a parallel structure accessor function.
update-function	Symbol. The name of an update function to be called whenever ***setf** is called on *accessor-function.*

RETURNED VALUE

update-function	Name of update function assigned.

SIDE EFFECTS

Assigns *update-function* as function to be called whenever ***setf** is called on *accessor-function*.

DESCRIPTION

Defines the *update-function* used for a given *accessor-function* in a call to ***setf.**

EXAMPLES

```
(*defsetf 'get-pvar-value 'modify-pvar-value)
```

REFERENCES

See also the dictionary entry for the ***setf** macro.

The macro ***undefsetf** may be used to remove the assignment made by ***defsetf**. See the definition of ***undefsetf** for more information.

*defstruct [*Macro*]

Defines a structure pvar type.

SYNTAX

***defstruct** *structure–name*
&optional *documentation* **&rest** *slot–descriptors*

***defstruct** (*structure–name* **&rest** *options*)
&optional *documentation* **&rest** *slot–descriptors*

ARGUMENTS

structure–name Symbol. Name of structure type.

options Series of structure option specifiers, described below, that control naming conventions and structure inheritance. Each supplied *option* must be of the form
(**:keyword** **&rest** *values*)

documentation String. Documentation string for structure.

slot–descriptors At least one slot descriptor of the form
(*slot–name default–init* **&rest** *slot–options*)

The three components of the *slot–descriptors* argument are described below.

slot–name Symbol. Name of slot.

default–init Front-end value. Single default value for all elements of the slot. Spread to all processors by the function **!!** when a parallel structure object is created. If the **:cm–initial–value** or **:cm–uninitialized–p** slot options are specified, then this argument is ignored when a parallel structure object is created.

slot–options Series of slot option keyword/value pairs of the form
:keyword *value*

RETURNED VALUE

structure–name Returns name of structure type.

SIDE EFFECTS

Defines both front-end and parallel structure types, along with constructor, accessor, copying, and modification operations for both stucture types.

DESCRIPTION

The macro ***defstruct** defines structure pvar types in *Lisp. A call to ***defstruct** defines both a Common Lisp scalar structure type and a Connection Machine parallel structure type. Further, ***defstruct** defines both scalar and parallel constructor, accessor, and assignment operations for these new data types. This double functionality of ***defstruct** allows structures to be passed back and forth between the Connection Machine system and the front-end computer.

A call to ***defstruct** does the following:

- defines a front-end **defstruct** type *structure–name*, with slots corresponding to the *slot–descriptors* of the ***defstruct**
- defines a new pvar type, **(pvar** *structure–name***)**; pvars of this type can contain only elements of type *structure–name*
- defines a parallel constructor function **make–***structure–name***!!**, which creates pvars of type **(pvar** *structure–name***)**
- defines pvar accessors of the form *structure–name–slot–name***!!** that take a pvar argument of type **(pvar** *structure–name***)** and return a copy of the structure slot *slot–name* in parallel
- defines ***setf** methods for these pvar accessors to permit modification of the structure pvar slots
- defines a *Lisp predicate, *structure–name***–p!!** to test whether a pvar is a parallel structure of the newly defined type
- defines a sequence pvar copying operation **copy–***structure–name***!!**, that takes a pvar of type **(pvar** *structure–name***)** and returns a copy of it

- permits the operations **!!**, ***setf** of **pref**, **array-to-pvar**, **pvar-to-array**, **array-to-pvar-grid**, and **pvar-to-array-grid** to accept a front-end **defstruct** object as the value stored in a structure pvar of the corresponding type

Keyword options in the *options* list control slot properties and naming conventions that apply to the parallel structure type as a whole. The keywords that may be supplied in the *options* list are described below.

- **:conc-name**

  Symbol. Used instead of *structure-name* as the prefix of slot accessor functions. If this keyword is supplied with a value of **nil**, or with no value at all, no prefix is attached to slot accessor functions.

- **:cm-constructor**

  Symbol. Used as the name of the structure pvar constructor function instead of the default, **make-***structure-name***!!**.

- **:parallel-cm-predicate**

  Symbol. Used as the name of the structure pvar predicate instead of the default, *structure-name***-p!!**.

- **:include**

  Symbol. Names a structure pvar type previously defined by ***defstruct** that is to be included in the definition of the new structure pvar type.

- **:cm-uninitialized-p**

  Boolean value. If **t**, is equivalent to supplying the **:cm-uninitialized-p** slot option in every *slot-options* list of the ***defstruct** form. Has no effect if **nil**.

In addition, almost all structure option keywords permitted by the Common Lisp **defstruct** operator may be included in the *options* list. (See Chapter 19, "Structures," in *Common Lisp: The Language*) The values supplied for these keywords are passed directly on to **defstruct**, and therefore have their normal effect. The only keywords that are not allowed are **:type**, **:named**, and **:initial-offset**.

Each *slot-descriptor* argument describes one slot of the parallel structure type being defined. The *slot-name* is used to name the slot in both the parallel structure type and the front-end structure type.

The value of *default-init* for each slot must be a form that returns a valid front-end value conforming to the type of the slot, as specified by the **:type** slot option. This value is distributed to all processors, as if by the function **!!**. If either of the options **:cm-uninitialized-p** or **:cm-initial-value** is specified in the *slot-options* list, then the *default-init* argument for that slot is ignored and can be specified as **nil**.

Keyword options in the *slot–options* list of each slot control typing and initialization of that slot.

One keyword option, **:type**, *must* be specified for each slot.

- **:type**

  Type specifier. Specifies data type of structure slot, for both front-end structures and structure pvars. This argument must specify a Common Lisp data type that is also valid as a pvar element type. Slots may not be specified as either general or mutable.

All other permissible *slot–options* keywords are described below.

- **:cm–type**

  Type specifier. Specifies data type of structure pvar slots, allowing extra control of structure pvar data types. Overrides data type specified by **:type** argument, but must be of a compatible data type (i.e., a more specific definition of the same basic data type).

- **:cm–initial–value**

  *Lisp form. Evaluated when structure pvars are created to provide default value for this structure slot. If unspecified, structure slot is initialized using *default–init* argument.

- **:cm–uninitialized–p**

  Boolean value. If **t**, structure objects are created with this slot uninitialized. Has no effect if **nil**. It is an error to supply a value for **:cm–initial–value** if the **:cm– uninitialized–p** argument is **t**. It is also an error to attempt to access an uninitialized structure slot before a value has been stored into it.

- **:read–only**

  Boolean value. If **t**, indicates that the slot is not to be modified. Has no effect if **nil**. It is an error to try to modify a slot that has been declared as **:read–only.**

EXAMPLES

An example of a call to ***defstruct** is

```
(*defstruct elephant
   (wrinkles 30000 :type (unsigned-byte 16))
   (tusks t :type boolean))
```

This expression defines both the front-end structure type **elephant** and a parallel structure type of **(pvar elephant)**. The front-end structure type is automatically defined by a call to **defstruct** of the form

```
(defstruct elephant
   (wrinkles 30000 :type (unsigned-byte 16))
   (tusks t :type boolean))
```

which defines a set of construction, accessor, predicate, and copying functions for the front-end structure type. The call to ***defstruct** also defines a set of parallel construction, accessor, predicate, and copying functions, described below. A parallel structure construction function called **make-elephant!!** is defined to create pvars of type **(pvar elephant)**. For example, the expression

```
(*defvar jumbo!! (make-elephant!!     :wrinkles (!! 0)))
```

defines a variable **jumbo!!** that contains a pvar with a wrinkle-free, tuskless **elephant** in each processor.

Parallel slot accessor functions, **elephant-wrinkles!!** and **elephant-trunk!!**, are defined, each of which takes a single argument of type **(pvar elephant)** and returns a copy of the contents of the specified slot as a pvar. For example,

```
(elephant-wrinkles!!    jumbo!!)  <=>  (!! 0)
(elephant-tusks!!       jumbo!!)  <=>  t!!
```

Methods are defined for ***setf** so that these slots can be modified in parallel. For example, the expression

```
(*setf (elephant-wrinkles!! jumbo!!) (!! 4000))
```

modifies the value of the **wrinkles** slot of each **elephant** structure in **jumbo!!** so that every **elephant** is moderately wrinkled. Methods are also defined for ***setf** so that a single value of a structure pvar of type **(pvar elephant)** can be modified.

```
(*setf (pref jumbo!! 0)
   (make-elephant :wrinkles 4000 :tusks t))
```

A parallel structure predicate, **elephant-p!!**, is defined. This takes a single pvar argument and returns **t!!** if it is of type **(pvar elephant)**.

```
(elephant-p!! jumbo!!)  =>  t!!
```

Finally, a parallel structure copying function, **copy-elephant!!**, is defined. It takes a pvar of type **(pvar elephant)**, and returns a copy as a temporary pvar.

```
(*defvar jumbo-copy!!)
(*set jumbo-copy!! (copy-elephant!! jumbo!!))
```

NOTES

Language Note:

Structure pvar slot accessor functions return a copy of the structure slot. If it is necessary to obtain the actual contents of the slot rather than a copy (e.g., to pass a slot to a function that modifies the slot's contents), use the macro **alias!!** in combination with the slot accessor function. However, it is only necessary to use the **alias!!** operator in specific circumstances. See the definition of **alias!!** for more information on where and when it should be used.

Important: the ***setf** macro automatically accesses the actual value specified by a slot accessor, so it is unnecessary to use **alias!!** in combination with ***setf**. For example, the expression

```
(*setf (alias!! (elephant-wrinkles!! jumbo!!)) (!! 4000))
```

can be equivalently, and more efficiently, written as

```
(*setf (elephant-wrinkles!! jumbo!!) (!! 4000))
```

Usage Note:

It is an error for any two slots to have the same name. Also, if any slot is given a *slot–name* of **p**, the **p** slot accessor *structname*–**p** will be shadowed by the *structname* structure pvar predicate *structname*–**p**. To get around this, use the ***defstruct :conc–name** option with an argument such as *structname*–**get–slot**.

REFERENCES

For a more detailed discussion of the ***defstruct** macro and of structure pvars in general, along with more examples of the use of ***defstruct**, see Chapter 4, entitled "Structure Pvars," in the **Lisp Reference Manual Supplement* Version 5.0.

The ***defstruct** macro is a parallel version of the Common Lisp defstruct macro. For a discussion of **defstruct**, and of the use of structures in Common Lisp, see Chapter 19, "Structures," in *Common Lisp: The Language.*

*defun [*Macro*]

Defines a *Lisp operator that takes pvar arguments and/or returns a pvar value, and automatically resets the CM stack upon exiting.

Note: In most cases, you can (and should) use **defun** rather than ***defun**. The differences are presented below. Read this entry *completely* before using ***defun** to define *Lisp functions!

SYNTAX

***defun** *fn–name arg–list* **&optional** *declarations documentation* **&body** *body*

ARGUMENTS

fn–name	Symbol. Name of function.
arg–list	List of arguments. Identical to the arglist parameter of **defun**.
declarations	Optional type declaration forms.
documentation	Optional documentation strings.
body	*Lisp forms. Body of function.

RETURNED VALUE

fn–name	Symbol. Name of parallel function being defined.

SIDE EFFECTS

Defines both a macro named *fn–name* and a function with a symbol name derived from *fn–name*.

Note: Because *fn–name* is defined as a macro, not a function, you must use the *Lisp operators ***apply** and ***funcall** to apply and funcall *fn–name*, and there are other things to be aware of—see below for more information.

DESCRIPTION

In general, user-defined functions containing *Lisp expressions may be defined using the Common Lisp **defun** operator. However, temporary pvars created during execution of some user-defined *Lisp functions can cause *Lisp to run out of stack space. The *Lisp operator ***defun** should be used in place of **defun** to define such functions.

The ***defun** macro is analogous to the Common Lisp **defun** and can be used in place of it in defining a function that accepts pvar arguments or returns a pvar result. However, the ***defun** macro adds extra code to reset the CM stack when the function exits, thus deallocating any temporary pvars that have been created during execution of the function. For efficiency, the ***defun** macro should be used only to define functions that must reset the CM stack.

The *declarations* argument can be any number of *Lisp declaration forms. These forms can include, but are not limited to, type declarations for the arguments to the function being defined by ***defun**. The *documentation* argument may be any number of documentation strings for the function.

There are two cases where a user-defined function would have to reset the CM stack. One is where the function will be called outside of *Lisp operators, such as ***set** and ***when**, that automatically reset the *Lisp stack when they exit. Another is where the function will be used within a complicated *Lisp expression that causes *Lisp to run out of stack space.

There are four rules to use in determining which *Lisp operators clear the CM stack, and therefore where it may be necessary to use ***defun**:

- Operators defined by ***defun** always reset the CM stack. These operators are indicated, both in their Dictionary entries and in the table of contents, by the notation [**Defun*].
- All of the pvar pretty printing operators (**ppp**, **ppp–css**, etc.) reset the CM stack.
- The following macros reset the CM stack:

***all**	***and**	***apply**	***cond**	***case**	***decf**	
***ecase**	***funcall**	***if**	***integer–length**		***incf**	
***let**	***let***	***logand**	***logior**	***logxor**	***map**	
***max**	***min**	***or**	**pref**	***pset**	***set**	
***setf**	***sum**	***unless**	***when**	**with–css–saved**		***xor**

- Functions whose names end in **!!** do *not* reset the CM stack.

A heuristic to follow in deciding whether or not to use ***defun** to define a function is that a user-defined function that takes pvar arguments and does not return a pvar value (such as the **log-sum-pvar** example below) should be defined using ***defun**, because these functions will most likely be called outside of a form such as ***set** that takes care of resetting the stack. Conversely, a user-defined function that takes pvar arguments and *does* return a pvar value should *not* be defined with ***defun**, unless its use causes *Lisp to run out of stack space.

One can declare that a function has been defined by ***defun** with the ***proclaim** operator. This allows the Common Lisp compiler to see that the "function" defined by ***defun** is actually a macro. For example,

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

(defun bar (x) (foo x))

(*defun foo (x) (*sum x))
```

Without the call to ***proclaim**, when **bar** is compiled the call to **foo** is treated as a function call. When **foo** is defined with ***defun**, it is actually defined as a macro, so that the call to **foo** within **bar** will not execute properly. Declaring that **foo** will be defined by ***defun** prior to the definition of any function that calls **foo** allows Lisp to compile these functions properly.

EXAMPLES

A sample call to ***defun** is

```
(*defun simply-functional (x y z)
   "A quite simple function of three complex arguments."
   "Author: Dent"
   (declare (type single-complex-pvar x y z))
   (+!! x y z))
```

An example of a case where ***defun** is necessary is the expression

```
(let ((total 0))
   (dotimes (i limit)
      (setq total (log-sum-pvar (random!! (!! i))))))
```

If the function **log-sum-pvar** is defined by

```
(defun log-sum-pvar (pvar)
   (log (*sum pvar)))
```

and if the value of **limit** is very large, the expression above will run out of stack space. The problem is that the expression **(random!! (!! i))** creates a temporary pvar on the CM stack on each iteration. The function **log-sum-pvar** does not reset the stack when it exits, and neither does any operator surrounding it within the **dotimes** loop. As the loop repeats, new temporary pvars are created on the stack until the stack is exhausted.

A better definition is

```
(*defun log-sum-pvar (pvar)
   (log (*sum pvar)))
```

This adds code that resets the CM stack following each invocation of **log-sum-pvar**. If **log-sum-pvar** is defined in this way, the example will execute normally.

An example of a case where the use of ***defun** is *not* necessary, and is in fact inefficient, is the expression

```
(dotimes (i limit)
   (*set result-pvar (+!! result-pvar (pvalue (!! i)))))
```

If the function **pvalue** is defined using **defun**, as in

```
(defun pvalue (data-pvar)
   (expt!! data-pvar (random!! (!! 10))))
```

the CM stack will not be exhausted even if **limit** becomes very large. The reason is that, like many *Lisp macros, ***set** automatically resets the stack after its argument expressions have been evaluated. If, in the example above, the function **pvalue** was defined with ***defun**, then the function would waste time needlessly resetting the stack each time around the **dotimes** loop.

Another example of a case in which ***defun** may be necessary is

```
(*proclaim '(ftype (function (t t t t) single-float-pvar)
                  component!!))

(defun component!! (x y z w)
   (declare (type single-float-pvar x y z w))
   (+!! (*!! x y) (cos!! z) (sqrt!! w)))

(defun stack-hog (x)
   (*set x
     (+!! (component!! (!! 3.0) (!! 4.0) (!! 5.0) (!! 6.0))
          (component!! (!! 3.0) (!! 4.0) (!! 5.0) (!! 6.0))
          (component!! (!! 3.0) (!! 4.0) (!! 5.0) (!! 6.0))
          (component!! (!! 3.0) (!! 4.0) (!! 5.0) (!! 6.0)))))
```

A call to **stack-hog** results in a large number of temporary pvars being allocated. Each call to **component!!** allocates four temporary pvars, and the body of **component!!** generates one or more temporary pvars as it executes. None of these pvars are reclaimed until the ***set** form exits.

By defining **component!!** with ***defun**, rather than **defun**, any temporary pvars allocated during the evaluation of each **component!!** form are reclaimed when the form exits. These include temporary pvars allocated during evaluation of the function's arguments (i.e., the constant expressions **(!! 3.0)**, **(!! 4.0)**, etc., in the example above) and also any temporary pvars generated by the execution of the body of **component!!**.

By reclaiming the stack each time a call to **component!!** exits, the amount of stack space required in executing **stack–hog** is significantly reduced. If a user-defined function defined with **defun** is consistently causing an application to run out of stack space, then it should be redefined with ***defun**.

Important: By redefining a function with ***defun**, when the function has previously been defined by **defun**, the function is being redefined as a macro. All forms in which the function is called must therefore be recompiled.

An example of a case where it using ***defun** is *not* necessary is

```
(*defun pvalue (pvar)
  (expt!! pvar (random!! (!! 10))))
```

If **pvalue** is defined with ***defun** in this way, then the expression

```
(dotimes (i limit)
  (*set result-pvar (+!! result-pvar (pvalue (!! i)))))
```

will execute unnecessarily slowly. The ***set** macro automatically resets the stack when it exits, but because the **pvalue** function was defined with ***defun**, it will perform an extra, redundant stack reset operation each time around the loop. Redefining **pvalue** with **defun** will improve performance:

```
(defun pvalue (pvar)
  (expt!! pvar (random!! (!! 10))))
```

NOTES

Implementation Note:

A call to ***defun** performs two definitions. It defines both a macro named *fn–name* and a function with a symbol name derived from *fn–name*. The macro expands into a call to the function, with enclosing code that records the original state of the stack and ensures that the stack is reset when the function exits.

Usage Notes:

To undefine functions created with ***defun**, use the *Lisp operator **un*defun**.

To apply ***defun** functions to lists of arguments, use the *Lisp operators ***apply** and ***funcall**. It is an error to use the Common Lisp operators **apply** and **funcall** for these purposes.

The *Lisp tracing operations for ***defun** functions are ***trace** and ***untrace**. It is an error to use the Common Lisp operators **trace** and **untrace** to trace a function defined with ***defun**.

In the hardware version of *Lisp, ***defun** uses underlying support functions to deal with stack memory reclamation. These underlying functions require that a CM be attached and cold-booted, so ***defun** functions likewise will not execute properly unless CM hardware is attached and cold-booted.

Compiler Note:

If a ***defun** is referenced prior to its definition in a file, then the Lisp compiler will not recognize it as a macro call (as you might intend), but will instead treat it as a call to an ordinary function. The "external" operator defined by ***defun** is a macro rather than a function, so these calls will signal an error.

There is a special ***proclaim** declaration that can be used to avoid this problem. For example:

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

(*proclaim
   '(ftype (function (t t) (pvar single-float)) xyzzy-foo))

(*proclaim '(type single-float-pvar z)) (*defvar z)

(defun bar ()
   (*set z    (xyzzy-foo (!! 3.0) (!! 4.0))))

(*defun xyzzy-foo (a b)
   (declare (type single-float-pvar a b))
   (+!! a b))
```

The ***proclaim** form declaring that a function is a ***defun** must be placed in the file prior to all references to that function, including its definition. In essence, the ***proclaim** form "forward references" the ***defun** definition, informing the compiler that a function will eventually be defined by ***defun**.

Important: Any type declarations for a ***defun** form must come after the **(*proclaim '(*defun ...))** form and before the actual ***defun** definition, as shown in the above example, or these declarations will not be used correctly.

*defvar

[Macro]

Allocates a new permanent pvar.

SYNTAX

***defvar** *pvar–name* **&optional** *initial–value–pvar documentation–string vp–set*

ARGUMENTS

pvar–name Symbol. Bound to newly allocated pvar.

initial–value–pvar Pvar expression. If supplied, used to initialize the values of the returned pvar.

documentation–string Optional documentation string.

vp–set VP set object. VP set to which the new pvar will belong. Defaults to the value of ***default–vp–set***.

RETURNED VALUE

pvar–name Returns *pvar–name*, the symbol to which the new pvar has been bound.

SIDE EFFECTS

Allocates a permanent pvar named *pvar–name* and binds it to the symbol *pvar–name*.

DESCRIPTION

This creates a new pvar that is permanently allocated. The *pvar–name* argument is a symbol that is bound globally to the allocated pvar. The optional argument *initial–value–pvar* may be any previously allocated pvar or pvar expression. The ***defvar** macro creates a new pvar, initializes it to the contents of *initial–value–pvar*, and binds *pvar–name* to that new pvar using **setq**. If no *initial–value–pvar* argument is given, the allocated pvar is uninitialized. During a ***cold–boot** operation, unless the

:undefine-all argument to ***cold-boot** has been specified as **t**, all pvars allocated by ***defvar** are reallocated and the supplied *initial-value-pvar* expression is reevaluated to reinitialize the pvars.

The optional argument *vp-set* defines the VP set to which the newly created pvar belongs. It defaults to the value of ***default-vp-set***.

The ***defvar** operator is intended to be used only at top level. It is an error to call ***defvar** from within a user-defined function, as in

```
(defun wrong-use-of-*defvar (x)
    (*defvar pvar (!! x))
    (*defvar pvar-squared (!! (* x x))))
```

The *Lisp operator **allocate!!** should be used instead to dynamically allocate global pvars from within a user-defined function. See the definition of **allocate!!** for more information.

EXAMPLES

The ***defvar** macro may be used to create a pvar with a specific initial value, as in

```
(*defvar pi!! (!! 3.14159265))
```

or with a value that is the result of a calculation, as in

```
(defparameter upper-bound 65536)
(*defvar limit-pvar (-!! (!! upper-bound) (self-address!!)))
```

The ***defvar** macro may also be used to create a pvar with no initial value, into which a value will later be stored by a call to an operator such as ***set**:

```
(*defvar scratch-pvar)

(*set scratch-pvar (/!! (1+!! (self-address!!))))
```

Note that it is an error to access the contents of a pvar defined in this way until an operator such as ***set** has been used to store a value into the pvar.

Array pvars and structure pvars may be created by a call to ***defvar**. However, when allocating either of these pvar types using ***defvar**, it is advisable to declare the type of pvar with ***proclaim**. Undeclared pvars into which any other type of data has been stored cannot be used to hold arrays or structures. For example,

```
(*defvar x)
(*set x (!! 3))
(*set x (!! #(1 2 3)))  ;;; This operation is not allowed
```

The ***defvar** macro can be used to create an array pvar in two ways: by directly creating the array pvar on the CM with a function such as **make–array!!**, as in

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

(ppp (aref!! fum (!! 1) (!! 2) (!! 0)) :end 10)
#\L #\L #\L #\L #\L #\L #\L #\L #\L #\L
```

or by simply using the **!!** operator to copy a front-end array into all processors, as in

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

Likewise, structure pvars can be defined by ***defvar** in two ways: by use of the parallel constructor function defined by ***defstruct**, for instance

```
(*defstruct elephant
   (wrinkles 30000 :type (unsigned-byte 16))
   (tusks t :type boolean))

(*proclaim '(type (pvar elephant) jumbo!!))
(*defvar jumbo!! (make-elephant!! :wrinkles (!! 300)
                                  :tusks t!!))

(*proclaim '(type (pvar elephant) jumbo-copy!!))
(*defvar jumbo-copy!! jumbo!!)
```

or by using **!!** to copy a front-end structure of a type defined by ***defstruct** to all processors, as in

```
(*defvar white-elephant-pvar
         (!! (make-elephant :wrinkles 0 :tusks nil)))
```

The *vp–set* argument can be used to specify the VP set to which the newly created pvar belongs. For example,

```
(def-vp-set ptbarnum '(128 128))

(*defvar ptbarnum-jumbo (!! 4.0) "Weight in tons" ptbarnum)
```

defines a VP set named **ptbarnum**, and a permanent pvar associated with **ptbarnum** named **ptbarnum–jumbo**.

The **def-vp-set** operator provides a way to lexically associate the definitions of permanent pvars with the definition of the VP set to which they belong. See the definition of **def-vp-set** for more information.

NOTES

Language Note:

Both permanent pvars and global pvars are allocated on the CM heap. Permanent pvars are allocated by ***defvar** and must be deallocated by the function ***deallocate-*defvars**. In contrast, global pvars are allocated by **allocate!!** and must be deallocated with ***deallocate.**

Style Note:

It is a good idea not to provide an *initial-value-pvar* argument to ***defvar** that is complex or dependant on global variables for its value. In these cases, reevaluation of the initialization form when the pvar is reallocated by ***cold-boot** may cause an error.

For example, the code fragment

```
(*cold-boot :initial-dimensions '(128 128))
(setq image-or-nil
   (make-image-array :dimensions '(128 128)))
(*defvar image!!
   (array-to-pvar-grid image-or-nil nil
                       :grid-end '(128 128)))
(setq image-or-nil nil)
(*cold-boot)  ;;; Error signalled in redefinition
```

signals an error on the second invocation of ***cold-boot** because *Lisp tries to reallocate **image!!** using the variable **image-or-nil**, which has been set to **nil.**

A better way to define pvars of this type is to use ***defvar** to declare the pvar, *without* an *initial-value-pvar* argument. The ***set** operator can then be used within an initialization routine to specify the value of the pvar, as in the following example:

```
(*defvar data-pvar)
(defun initialize-pvars ()
   (*set data-pvar
      (complicated-operation-returning-data-pvar)))
```

REFERENCES

See also the pvar allocation and deallocation operations

allocate!!	**array!!**	
***deallocate**	***deallocate–*defvars**	
front–end!!	***let**	***let***
make–array!!	**typed–vector!!**	**vector!!**
!!		

See also the *Lisp predicate **allocated–pvar–p**.

See the *Lisp glossary for definitions of the different kinds of pvars that are allocated on the CM stack and heap.

See Chapter 4, "*Lisp Types and Declaration," for more information about pvar types, type coercion, and undeclared pvars.

def–vp–set [*Macro*]

Defines a permanent VP set object, possibly with associated pvars.

SYNTAX

def–vp–set *vp–set–name vp–set–dimensions*
&key :geometry–definition–form :*defvars

ARGUMENTS

vp–set–name	Symbol. Name of VP set to which VP set object is bound.
vp–set–dimensions	List of integers or **nil**. Defines dimensions of VP set.
:geometry–definition–form	Geometry object or **nil**. Defines geometry of VP set.
:*defvars	List of pvar specifiers. Defines pvars created with ***defvar** that are associated with the new VP set.

RETURNED VALUE

vp–set–name	Symbol. Name of newly defined VP set.

SIDE EFFECTS

Creates a VP set and binds it to the symbol *vp–set–name*. Defines all pvars specified by the **:*defvars** keyword argument by using ***defvar**.

DESCRIPTION

The **def–vp–set** macro defines a permanent VP set named *vp–set–name* and should be used only at top level. Unless the user explicitly specifies that they should be deallocated, permanent VP sets and the pvars associated with them are automatically reallocated during a ***cold–boot** operation. The **def–vp–set** macro does *not* alter the value of ***current–vp–set***. Use the **set–vp–set** or ***with–vp–set** operators to change the current VP set.

The **def–vp–set** macro returns the symbol *vp–set–name*, after binding it to a VP set object with the specified *vp–set–dimensions* and associated **:*defvars**.

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, where *n* is the length of the list of integers supplied.

Each dimension must be a power of two. The product of all dimensions must be equal to either the physical machine size or a power-of-two multiple of 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, returns a geometry object. 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 the definition of **create–geometry** for a description of geometry objects.

If either *vp–set–dimensions* or a **:geometry–definition–form** is supplied, the VP set *vp–set–name* is created as a *fixed-size* VP set; its geometry is fixed and does not change. The returned VP set is initialized and allocated at ***cold–boot** time. If either *vp–set–dimensions* or a **: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 both *vp–set–dimensions* and the **:geometry–definition–form** argument are **nil**, then the returned VP set is defined as a *flexible* VP set. This type of VP set has no specific geometry until it has been *instantiated* by calling the function **allocate–processors–for–vp–set** or **with–processors–allocated–for–vp–set**. This may be done any time after a call has been made to ***cold–boot**.

The keyword **:*defvars** takes a list of lists, each of which specifies a permanent pvar that is associated with the VP set *vp–set–name*. Each sublist must be of the form

(*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 only when the VP set *set–name* is instantiated. Such pvars are reallocated and reinitialized by ***cold–boot.**

EXAMPLES

This expression creates a three-dimensional VP set named **fred** with dimensions 1024 by 32 by 128.

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

This expression creates a two-dimensional VP set named **george** with a VP ratio of 32, i.e., thirty-two virtual processors for each physical processor attached.

```
(def-vp-set george (list *minimum-size-for-vp-set* 32))
```

The expression

```
(def-vp-set anne '(65536)
    :*defvars ((x (!! 1) nil (field-pvar 2))
               (y (self-address!!))))
```

creates a one-dimensional VP set named **anne**, and defines two permanent pvars associated with **anne** as if by the following forms:

```
(def-vp-set anne '(65536))
(*proclaim '(type (field-pvar 2) x))
(*defvar x (!! 1) nil anne)
(*defvar y (self-address!!) nil anne)
```

If the arguments *vp–set–dimensions* and **:geometry–definition–form** are both **nil**, then a VP set with no initial geometry, known as a *flexible VP set*, is defined. Flexible VP sets must be instantiated before use, by either of the instantiation operators **allocate–processors–for–vp–set** or **with–processors–allocated–for–vp–set**. For example, the pair of expressions

```
(def-vp-set gumby nil)
(allocate-processors-for-vp-set gumby '(128 64 32))
```

defines a flexible VP set named **gumby**, and instantiates **gumby** as a three-dimensional VP set. The expression

```
(deallocate-processors-for-vp-set gumby)
```

deinstantiates **gumby**, so that it may be instantiated with a different number of processors. The expression

```
(with-processors-allocated-for-vp-set gumby
   :dimensions '(128 64 32)
   (user-defined-function))
```

performs the same instantiation and deinstantiation automatically, temporarily instantiating **gumby** during the execution of the **user–defined–function.**

NOTES

Because the newly created VP set object is simply bound as the value of the symbol *vp–set–name*, it is a good idea to choose a *vp–set–name* that will *not* be used as the name of a global variable. For example, if the expressions

```
(def-vp-set data-set '(512 512))
```

and

```
(*defvar data-set (random!! (self-address!!)))
```

are evaluated in order, the permanent pvar created by ***defvar** will replace the VP set created by **def–vp–set** as the value of the symbol **data–set.**

REFERENCES

See the *Lisp glossary for definitions of permanent, temporary, fixed-size, and flexible VP sets.

See also the following VP set definition and deallocation operators:

create–vp–set **let–vp–set**
deallocate–def–vp–sets **deallocate–vp–set**

See also the following geometry definition operator:

create–geometry

The following math utilities are useful in defining the size of VP sets:

next–power–of–two–>= **power–of–two–p**

See also the following flexible VP set operators:

allocate–vp–set–processors	**allocate–processors–for–vp–set**
deallocate–vp–set–processors	**deallocate–processors–for–vp–set**
set–vp–set–geometry	**with–processors–allocated–for–vp–set**

These operations are used to select the current VP set:

set–vp–set	***with–vp–set**

See also the following VP set information operations:

dimension–size	**dimension–address–length**
describe–vp–set	**vp–set–deallocated–p**
vp–set–dimensions	**vp–set–rank**
vp–set–total–size	**vp–set–vp–ratio**

delete–initialization *[Function]*

Removes *Lisp code placed on initialization lists by **add–initialization.**

SYNTAX

delete–initialization *name–of–form init–list–name*

ARGUMENTS

name–of–form	Character string. Name of initialization form to remove.
init–list–name	Symbol or list of symbols. Initialization list(s) from which the specified initialization form is removed.

RETURNED VALUE

nil	Executed for side effect.

SIDE EFFECTS

The named initialization form is removed from the initialization list or lists specified by *init–list–name.*

DESCRIPTION

The function **delete–initialization** removes a named initialization from one or more of the following *Lisp initialization lists:

- ***before–*cold–boot–initializations***
  *Lisp code evaluated immediately prior to any call to ***cold–boot.**
- ***after–*cold–boot–initializations***
  *Lisp code evaluated immediately after to any call to ***cold–boot.**
- ***before–*warm–boot–initializations***
  *Lisp code evaluated immediately prior to any call to ***warm–boot.**

- ***after–*warm–boot–initializations***
  *Lisp code evaluated immediately after to any call to ***warm–boot**.

The arguments are specified in the same manner as the first and third arguments for **add–initialization**.

EXAMPLES

The function **delete–initialization** is the recommended way to remove initializations from the above lists. For example, the expression

```
(add-initialization "Recompute Important Pvars"
   '(recompute-important-pvars *number-of-processors-limit*)
   '*after-*cold-boot-initializations*)
```

adds an initialization form named "Recompute Important Pvars" to the list ***after–*cold–boot–initializations***. Evaluating the expression

```
(delete-initialization "Recompute Important Pvars"
          '*after-*warm-boot-initializations*)
```

will remove the initialization form.

REFERENCES

See also the related operation **add–initialization**.

See also the following Connection Machine initialization operators:

***cold–boot** ***warm–boot**

See also the character attribute initialization operator **initialize–character**.

deposit-byte!! [*Function*]

Performs a parallel byte deposit operation on the supplied pvars.

SYNTAX

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

ARGUMENTS

into-pvar	Integer pvar. Integer into which byte is deposited.
position-pvar	Integer pvar. Bit position, zero-based, at which value of *byte-pvar* is deposited.
size-pvar	Integer pvar. Bit size of byte to deposit.
byte-pvar	Integer pvar. Byte to deposit into *into-pvar*.

RETURNED VALUE

newbyte-pvar	Temporary integer pvar. In each active processor, contains a copy of *into-pvar* with *size-pvar* bits beginning at *position-pvar* replaced by low-order bits of *byte-pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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

When the *into-pvar* is positive, zeros are appended as high order bits of *byte-pvar* as needed. When the *into-pvar* is negative, ones are appended as high order bits of *byte-pvar* as needed.

EXAMPLES

The returned value may have more bits than *into–pvar* if the inserted field extends beyond the most significant bit of *into–pvar*. For example,

```
(deposit-byte!! (!! #B11) (!! 1) (!! 2) (!! #B10))
```

returns

```
(!! 5)  <=>  (!! #B101)
```

NOTES

Usage note:

This function is especially fast when both *position–pvar* and *size–pvar* are constants, as in **(!!** *positive–integer***)**.

REFERENCES

See also these related byte manipulation operators:

byte!!	**byte–position!!**	**byte–size!!**
deposit–field!!	**dpb!!**	
ldb!!	**ldb–test!!**	**load–byte!!**
mask–field!!		

deposit–field!! *[Function]*

Performs a parallel bit field copy operation on the supplied pvars.

SYNTAX

deposit–field!! *into–pvar bytespec–pvar integer–pvar*

ARGUMENTS

into–pvar	Integer pvar. Integer into which bit field is copied.
bytespec–pvar	Byte specifier pvar, as returned from **byte!!**. Determines position and size of byte in *into–pvar* which is replaced.
integer–pvar	Integer pvar. Integer from which bit field is copied.

RETURNED VALUE

newbyte–pvar	Temporary integer pvar. In each active processor, contains a copy of *into–pvar* with *size–pvar* bits beginning at *position–pvar* replaced by the corresponding bits of *integer–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The function **deposit–field!!** is the parallel equivalent of the Common Lisp function **deposit–field**. The *newbyte–pvar* result contains, for each processor, a copy of the value of *into–pvar* with the byte specified by *bytespec–pvar* replaced by the corresponding bits of of *integer–pvar*. The result therefore agrees with *integer–pvar* in the byte specified, and with the original value of *newbyte–pvar* everywhere else.

EXAMPLES

```
(deposit-field newbyte-pvar  (byte!! size-pvar position-pvar)
                              integer-pvar)
<=>
(dpb!! (ldb!! (byte!! size-pvar position-pvar) newbyte-pvar)
       (byte!! size-pvar position-pvar) integer-pvar)
```

REFERENCES

See also these related byte manipulation operators:

byte!!	**byte-position!!**	**byte-size!!**
deposit-byte!!	**dpb!!**	
ldb!!	**ldb-test!!**	**load-byte!!**
mask-field!!		

describe–pvar [*Function*]

Displays information about a pvar.

SYNTAX

describe–pvar *pvar* **&optional** *stream*

ARGUMENTS

pvar	Pvar expression. Pvar to describe.
stream	Stream object. Defaults to ***standard–output***.

RETURNED VALUE

nil	Evaluated for side effect only.

SIDE EFFECTS

Prints formatted description of *pvar* to *stream*.

DESCRIPTION

This function prints out information about *pvar* in a neat format. The printed information includes memory location, field ID, length, type, and VP set of the *pvar*.

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
```

REFERENCES

See also the following general pvar information operators:

allocated–pvar–p	**pvar–exponent–length**	
pvar–length	**pvar–location**	**pvar–mantissa–length**
pvar–name	**pvarp**	**pvar–plist**
pvar–type	**pvar–vp–set**	

describe–vp–set *[Function]*

Displays information about a VP set.

SYNTAX

describe–vp–set *vp–set* **&key :*defvars :verbose :stream**

ARGUMENTS

vp–set	VP set object. VP set to be described.
:*defvars	Boolean value. Determines whether pvars associated with the specified VP set are described. Defaults to **t**.
:verbose	Boolean value. Determines whether to display detailed information about the VP set. Defaults to **nil**.
:stream	A stream. Defaults to ***standard–output***. Stream to which output is printed.

RETURNED VALUE

nil	Evaluated for side effect only.

SIDE EFFECTS

Prints formatted description of *vp–set* to the ***standard–output*** stream. If **:*defvars** argument is **t**, displays information about each pvar associated with *vp–set*.

DESCRIPTION

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.

The argument *vp–set* must be a temporary or permanent 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 **: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.

EXAMPLES

A sample call to **describe–vp–set** is 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*))
```

In the example above, ***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*))
```

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.

REFERENCES

See also the following VP set information operations:

dimension–size	**dimension–address–length**
vp–set–deallocated–p	
vp–set–dimensions	**vp–set–rank**
vp–set–total–size	**vp–set–vp–ratio**

digit–char!! *[Function]*

Performs a parallel conversion from integer digits to characters.

SYNTAX

digit–char!! *digit–pvar* **&optional** *radix–pvar font–pvar*

ARGUMENTS

digit–pvar	Integer pvar. Numeric value to construct as a character.
radix–pvar	Integer pvar. Radix for which to construct character. Defaults to **(!! 10)**.
font–pvar	Integer pvar. Font attribute for newly constructed character. Defaults to **(!! 0)**.

RETURNED VALUE

char–pvar	Temporary character pvar. In each active processor, contains a character in the font specified by *font–pvar* which is the digit representation of *digit–pvar* in the radix specified by *radix–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function attempts to construct a character pvar containing, in each processor, a character of font *font–pvar* representing the value of *digit–pvar* in radix *radix–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 *digit-pvar* is less than *radix-pvar*.

Characters returned by **digit-char!!** are always in upper case.

EXAMPLES

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

REFERENCES

See also the related character/integer pvar conversion operators:

char-code!!	**char-int!!**	**code-char!!**
digit-char!!	**int-char!!**	

digit–char–p!! *[Function]*

Performs a parallel test for digit characters on the supplied pvar.

SYNTAX

digit–char–p!! *character–pvar* **&optional** *radix–pvar*

ARGUMENTS

character–pvar	Character pvar. Pvar to be tested for digit characters.
radix–pvar	Integer pvar. Determines radix of digit characters that are accepted as valid.

RETURNED VALUE

digit–charp–pvar	Temporary pvar. Contains the numeric value of *character–pvar*, where *character–pvar* contains a valid digit character in the radix *radix–pvar*, in each active processor. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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

In each processor where *character–pvar* contains a character that is a digit in the base specified by *radix–pvar*, **digit–char–p!!** returns a non-negative integer indicating the numeric value of the digit. In those processors where the elements of *character–pvar* are not digits of the specified radix, **digit–char–p!!** returns **nil**.

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)**.

EXAMPLES

```
(digit-char-p!! (!! #\3))  <=>  (!! 3)
```

NOTES

Language Note:

Digit character pvars are always graphic character pvars.

dimension–address–length [*Function*]

Returns the number of bits necessary to represent a NEWS address coordinate for the specified dimension in the current VP set.

SYNTAX

dimension–address–length *dimension*

ARGUMENTS

dimension Integer. Zero-based number of dimension for which address length is returned.

RETURNED VALUE

bit–length Integer. Number of bits needed to represent a NEWS address coordinate for *dimension* in the current VP set.

SIDE EFFECTS

None.

DESCRIPTION

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

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

EXAMPLES

If the value of ***current-cm-configuration*** is **(32 16)**, then

```
(dimension-address-length 0) => 5
```

REFERENCES

See also the following VP set information operations:

dimension-size

describe-vp-set	**vp-set-deallocated-p**
vp-set-dimensions	**vp-set-rank**
vp-set-total-size	**vp-set-vp-ratio**

dimension–size

[Function]

Returns the size of the specified dimension of the current VP set.

SYNTAX

dimension–size *dimension*

ARGUMENTS

dimension	Integer. Zero-based number of dimension for which the size is returned.

RETURNED VALUE

dimension–size	Integer. Size of specified dimension in current VP set.

SIDE EFFECTS

None.

DESCRIPTION

This function returns the size of the specified *dimension* of the current VP set.

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

EXAMPLES

If the value of ***current–cm–configuration*** is **(32 16)**, then

```
(dimension-size 0) => 32
```

REFERENCES

See also the following VP set information operations:

dimension–address–length

describe–vp–set	**vp–set–deallocated–p**
vp–set–dimensions	**vp–set–rank**
vp–set–total–size	**vp–set–vp–ratio**

do–for–selected–processors [*Macro*]

Iteratively binds a symbol to the send address of each active processor while executing the body of the form.

SYNTAX

do–for–selected–processors (*symbol*) **&body** *body*

ARGUMENTS

symbol	Symbol. Bound to the send address of each active processor.
body	*Lisp forms. Evaluated for each active processor.

RETURNED VALUE

nil	Normally returns **nil**, unless a non-local exit operator such as **return** is used within the *body*.

SIDE EFFECTS

None other than those produced by the forms in *body*.

DESCRIPTION

This form evaluates *body* as many times as there are active processors in the currently selected set. On each iteration, *symbol* bound to the send address of a different active processor.

EXAMPLES

Using **do–for–selected–processors**, the function **list–of–active–processors** could be written as

```
(defun my-list-of-active-processors ()
   (let ((result nil))
      (do-for-selected-processors (proc)
         (push proc result))
      (nreverse result)))
```

NOTES

As with the Common Lisp **dotimes**, the **return** function may be used to exit the **do–for–selected–processors** form immediately, returning a value. Normally, **do–for–selected–processors** returns **nil.**

Also, remember that while the supplied *body* forms are evaluated once for each active processor, each loop is evaluated in the currently selected set, so that all parallel operations are performed only by active processors. If you want the *body* to be executed by all processors, include a call to ***all**, as in:

```
(do-for-selected-processors (proc)
   (*all
      body-forms... ))
```

REFERENCES

See also the related operation **list–of–active–processors.**

See also the related processor selection operators

***all**	
***if**	**if!!**
***case**	**case!!**
***cond**	**cond!!**
***ecase**	**ecase!!**
***unless**	***when**
with–css–saved	

dot–product [*Function*]

Returns the dot product of two front-end vectors.

SYNTAX

dot–product *vector1 vector2*

ARGUMENTS

vector1, vector2 Front-end vectors for which the dot product is returned.

RETURNED VALUE

dot–prod–vector Front-end vector. Dot product of *vector1* and *vector2*.

SIDE EFFECTS

None.

DESCRIPTION

This is the front-end equivalent of **dot–product!!**.

EXAMPLES

```
(dot-product #(1.0 2.0 3.0) #(4.0 5.0 6.0))  =>  32.0
```

NOTES

For those not familiar with dot products, the dot product of two vectors

$(x_1, x_2, x_3, ...x_n)$ and $(y_1, y_2, y_3, ...y_n)$

is $(x_1 * y_1) + (x_2 * y_2) + (x_3 * y_3) + ... + (x_n * y_n)$

The **dot–product** operation returns the dot product of *vector1* and *vector2.*

REFERENCES

This function is one of a number of front-end vector operators, listed below:

cross–product	**dot–product**	**v+ v– v* v/**	
v+–constant	**v—constant**	**v*–constant**	**v/–constant**
vabs	**vabs–squared**	**vceiling**	**vector–normal**
vfloor	**vround**	**vscale**	
vscale–to–unit–vector		**vtruncate**	

These functions are the serial equivalents of the corresponding vector pvar operations. See Chapter 1, "*Lisp Overview," of this Dictionary for a list of these functions.

dot–product!! [*Function*]

Performs a parallel dot product operation on the supplied vector pvars.

SYNTAX

dot–product!! *vector–pvar1 vector–pvar2*

ARGUMENTS

vector–pvar1, vector–pvar2
Vector pvars, for which the dot product is returned. Both vector pvars must have the same number of elements.

RETURNED VALUE

dot–prod–pvar Temporary pvar. In each active processor, contains the dot product of the corresponding values of *vector–pvar1* and *vector–pvar2*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function returns a scalar pvar of the proper type and size. In each processor, the inner product of the two vectors is returned.

The following forms are equivalent:

```
(dot-product!! c1-pvar c2-pvar)
<=>
(reduce!! #'+!! (amap!! #'*!! c1-pvar c2-pvar))
<=>
(*let ((result (!! 0)))
   (dotimes (j (*array-total-size (c1-pvar)))
      (*incf result (*!!(aref!! c1-pvar (!! j))
                         (aref!! c2-pvar (!! j)))))
   result))
```

EXAMPLES

```
(dot-product!!(!! #(1.0 2.0 3.0))
              (!! #(4.0 5.0 6.0)))
<=>
(!! 32.0)
```

REFERENCES

This function is one of a number of related vector pvar operators, listed below:

cross–product!!	**dot–product!!**	**v+!! v–!! v*!! v/!!**	
v+scalar!!	**v–scalar!!**	**v*scalar!!**	**v/scalar!!**
vabs!!	**vabs–squared!!**	**vector–normal!!**	**vscale!!**
vscale–to–unit–vector!!		***vset–components**	

dpb!! [Function]

Performs a parallel byte deposit operation on the supplied pvars.

SYNTAX

dpb!! *byte–pvar bytespec–pvar into–pvar*

ARGUMENTS

byte–pvar	Integer pvar. Byte to deposit.
bytespec–pvar	Byte specifier pvar, as returned by **byte!!**. Determines position and size of the byte that is replaced in *into–pvar*.
into–pvar	Integer pvar. Integer into which byte is deposited.

RETURNED VALUE

newbyte–pvar	Temporary integer pvar. In each active processor, contains a copy of *into–pvar* with the byte specified by *bytespec–pvar* replaced by the value of *byte–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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

The function **dpb!!** returns an integer pvar that is a copy of *into–pvar* with the byte specified by *bytespec–pvar* replaced by the corresponding byte from *byte–pvar*.

The following forms are equivalent:

```
(deposit-byte!! integer-pvar pos-pvar size-pvar newbyte-pvar)
<=>
(dpb!!
  newbyte-pvar (byte!! size-pvar position-pvar) integer-pvar)
```

REFERENCES

See also these related byte manipulation operators:

byte!!	**byte–position!!**	**byte–size!!**
deposit–byte!!	**deposit–field!!**	
ldb!!	**ldb–test!!**	**load–byte!!**
mask–field!!		

*ecase, ecase!! [*Macro*]

Evaluates *Lisp forms with the currently selected set bound according to the value of a pvar expression.

SYNTAX

***ecase/ecase!!** *value–expression* (*key–expression–1* **&rest** *body–forms–1*)
(*key–expression–2* **&rest** *body–forms–2*)
...
(*key–expression–n* **&rest** *body–forms–n*)

ARGUMENTS

value–expression	Pvar expression. Value to compare against *key–expression–n* in each clause.
key–expression–n	Scalar expression. Evaluated, compared with *value–expression.* Selects processors in which to perform the corresponding *body–forms*. May also be a list of such expressions, in which case each expression is compared with *value–expression.*
body–forms–n	*Lisp forms. These forms are evaluated with the currently selected set restricted to those processors in which *value–expression* is **eql!!** to **(!!** *key–expression–n* **)**.

RETURNED VALUE

For ***ecase**:

nil	Evaluated for side effect only.

For **ecase!!**:

case–value–pvar	Temporary pvar. In each active processor, contains the value returned by *body–forms–n* if and only if *value–expression* is **eql** to *key–expression–n.*

SIDE EFFECTS

For ***ecase**:

None aside from those of the individual *body-forms*.

For **ecase!!**:

The returned pvar is allocated on the stack.

DESCRIPTION

The ***ecase** and **ecase!!** macros are parallel equivalents of the Common Lisp **ecase** operation. The two operators each select groups of processors to execute different portions of *Lisp code. Unlike **ecase**, however, ***ecase** and **ecase!!** evaluate all clauses.

The main difference between ***ecase** and **ecase!!** is that ***ecase** is used only for the side effects of its body forms, while **ecase!!** also constructs and returns a value-pvar that contains the value returned by its *body-forms*. Both ***ecase** and **ecase!!** signal an error if any active processors do not evaluate one of the supplied clauses.

EXAMPLES

When the following forms are evaluated,

```
(*defvar result (!! 1))
(*ecase (mod!! (self-address!!) (!! 4))
   (0     (*set result (!! 0)))
   ((1 2) (*set result (self-address!!)))
   (3     (*set result (!! -1))))
```

result is bound to a pvar with the values 0, 1, 2, –1, 0, 5, 6, –1, etc.

Similarly, when

```
(ecase!! (mod!! (self-address!!) (!! 4))
   (0     (!! 0))
   ((1 2) (self-address!!))
   (3 (!! -1)))
```

is executed, the returned pvar contains the values 0, 1, 2, –1, 0, 5, 6, –1, etc.

NOTES

Usage Notes:

It is an error for two ***ecase** or **ecase!!** clauses to contain the same *key–expression.* If two **ecase!!** clauses contain the same key, the returned pvar contains the values returned by the body forms in the first of the clauses.

Forms such as **throw**, **return**, **return–from**, and **go** may be used to exit a block or looping construct from within a processor selection operator such as ***ecase** or **ecase!!**. However, doing so will leave the currently selected set in the state it was in at the time the non-local exit form is executed. To avoid this, use the *Lisp macro **with–css–saved**.

See the dictionary entry for **with–css–saved** for more information.

Performance Note:

Currently, ***ecase** and **ecase!!** clauses execute serially, in the order in which they are supplied. At any given time, therefore, the number of processors active within a ***ecase** or **ecase!!** clause is a subset of the currently selected set at the time the ***ecase** or **ecase!!** form was entered. Providing a large number of clauses therefore can result in inefficient processor usage.

REFERENCES

See also the related operators

***all**	***cond**	**cond!!**	***case**	**case!!**
***if**	**if!!**	***unless**	***when**	**with–css–saved**

enumerate!! *[Function]*

Returns a pvar with a unique integer in each active processor.

SYNTAX

enumerate!!

ARGUMENTS

Takes no arguments.

RETURNED VALUE

enumerated–pvar Temporary pvar. Contains a unique integer value in each active processor.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function returns a pvar that contains, in each active processor, a unique number from 0 up to one less than the number of selected processors. The numbers are ordered, with 0 placed in the processor with the smallest send address, 1 placed in the processor with the next smallest send address, and so on.

```
(enumerate!!)  <=>  (1-!! (scan!! (!! 1) '+!!))
```

EXAMPLES

The **enumerate!!** function enumerates active processors. For example, the expression

```
(ppp  (if!!  (oddp!! (self-address!!))
                    (enumerate!!) (!! 99))
      :end 10)
```

displays the following values

```
99 0 99 1 99 2 99 3 99 4
```

Note that only the odd processors (those selected by the **(oddp!! (self-address!!))** test form) are enumerated.

The **enumerate!!** function is often used to pack values in active processors into the first *n* processors, where *n* is the number of active processors. For example

```
(*defvar pvar-to-be-packed (random!! (!! 10)))
(ppp pvar-to-be-packed :end 10)
8 3 1 9 2 2 1 4 3 1

(*defvar packed-pvar (!! 0))
(*when (evenp!! (self-address!!))
  (*pset :no-collisions pvar-to-be-packed packed-pvar
         (enumerate!!)))
(ppp packed-pvar :end 5)
8 1 2 1 3
```

The values in the active (even) processors are packed into the first *n*/2 processors.

NOTES

If all processors in the CM are selected, **enumerate!!** is equivalent to the function **self-address!!**. However, in this case, calling **self-address!!** itself is much more efficient.

REFERENCES

See also the related functions

rank!!	**self!!**	
self-address!!	**self-address-grid!!**	**sort!!**

eq!! *[Function]*

Performs a parallel comparison of the supplied pvars for identical values.

SYNTAX

eq!! *pvar1 pvar2*

ARGUMENTS

pvar1, pvar2	Simple pvars. Compared in parallel for identical values.

RETURNED VALUE

eq-pvar	Temporary boolean pvar. In each active processor, contains the value **t** if *pvar1* and *pvar2* contain identical values. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This is the parallel equivalent of the Common Lisp function **eq**. It performs a parallel comparison of the supplied pvars for identical numeric and character values.

EXAMPLES

```
(eq!! (!! #\c) (!! #\c))  <=>  t!!
```

NOTES

Language Note:

There is no fundamental difference between the operations performed by the functions **eql!!** and **eq!!** in *Lisp. This differs from Common Lisp, where **eql** and **eq** are defined such that **eql** performs a less restrictive test than **eq**. Both **eq!!** and **eql!!** are included in *Lisp for readability, and programmers should use the test that most clearly indicates the type of comparison being performed.

eql!! [*Function*]

Performs a parallel comparison of the supplied pvars for identical values.

SYNTAX

eql!! *pvar1 pvar2*

ARGUMENTS

pvar1, pvar2	Simple pvars. Compared in parallel for identical values.

RETURNED VALUE

eql–pvar	Temporary boolean pvar. In each active processor, contains the value **t** if *pvar1* and *pvar2* contain identical values. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This is the parallel equivalent of the Common Lisp function **eql**. It performs a parallel comparison of the supplied pvars for identical values. Numbers of the same type and value are considered identical by **eql!!**, as are character objects that represent the same character.

EXAMPLES

```
(eql!! (!! #\c) (!! #\c))  <=>  t!!
```

NOTES

Language Note:

There is no fundamental difference between the operations performed by the functions **eql!!** and **eq!!** in *Lisp. This differs from Common Lisp, where **eql** and **eq** are defined such that **eql** performs a less restrictive test than **eq**. Both **eq!!** and **eql!!** are included in *Lisp for readability, and programmers should use the test that most clearly indicates the type of comparison being performed.

equal!! [*Function*]

Performs a parallel comparison of the supplied pvars for equality.

SYNTAX

equal!! *pvar1 pvar2*

ARGUMENTS

pvar1, pvar2 Pvars. Compared in parallel for equality.

RETURNED VALUE

equal-pvar Temporary boolean pvar. In each active processor, contains the value **t** if the values of *pvar1* and *pvar2* are equal.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function is equivalent to **eql!!** if *pvar1* and *pvar2* are boolean or character pvars. If *pvar1* and *pvar2* are numeric pvars, it is equivalent to **=!!**. If the parameters are bit-vectors or strings, **equalp!!** performs the appropriate elementwise comparison. Otherwise if the parameters are structure or array pvars, **equal!!** returns **nil!!**..

equalp!! [*Function*]

Performs a parallel comparison of the supplied pvars for equality.

SYNTAX

equalp!! *pvar1 pvar2*

ARGUMENTS

pvar1, pvar2	Pvars. Compared in parallel for equality.

RETURNED VALUE

equalp-pvar	Temporary boolean pvar. In each active processor, contains the value **t** if the values of *pvar1* and *pvar2* are equal.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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.

evenp!! [*Function*]

Performs a parallel test for even numeric values on the supplied pvar.

SYNTAX

evenp!! *integer–pvar*

ARGUMENTS

integer–pvar	Integer pvar. Pvar to be tested for even values.

RETURNED VALUE

evenp–pvar	Temporary boolean pvar. Contains the value t in each active processor where the corresponding value of *integer–pvar* is even.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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

EXAMPLES

```
(ppp (evenp!! (self-address!!)) :end 12)
```

displays

```
T NIL T NIL T NIL T NIL T NIL T NIL
```

---

every!! *[Function]*

Tests in parallel whether the supplied pvar predicate is true for every set of elements having the same indices in the supplied sequence pvars.

SYNTAX

every!! *predicate sequence–pvar* **&rest** *sequence–pvars*

ARGUMENTS

predicate — Boolean pvar predicate. Used to test elements of sequences in the *sequence–pvar* arguments. Must take as many arguments as the number of *sequence–pvar* arguments supplied.

sequence–pvar, sequence–pvars — Sequence pvars. Pvars containing, in each processor, sequences to be tested by *predicate*.

RETURNED VALUE

every–pvar — Temporary boolean pvar. Contains the value **t** in each active processor in which every set of elements having the same indices in the sequences of the *sequence–pvars* satisfies the *predicate*. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The **every!!** function returns a boolean pvar indicating in each processor whether the supplied *predicate* is true for every set of elements with the same indices in the sequences of the supplied *sequence–pvars*.

In each processor, the *predicate* is first applied to the index 0 elements of the sequences in the *sequence–pvars*, then to the index 1 elements, and so on. The *n*th time *predicate*

is called, it is applied to the *nth* element of each of the sequences. If *predicate* returns **nil** in any processor, that processor is temporarily removed from the currently selected set for the remainder of the operation. The operation continues until the shortest of the *sequence–pvars* is exhausted, or until no processors remain selected.

The pvar returned by **every!!** contains **t** in each processor where *predicate* returns the value **t** for every set of sequence elements. If *predicate* returns **nil** for any set of sequence elements in a given processor, **every!!** returns **nil** in that processor.

EXAMPLES

```
(every!! 'equalp!! (!! #(1 2 3)) (!! #(1 2 3)))  <=>  t!!
(every!! '<!! (!! #(1 2 3)) (!! #(2 3 0)))  <=>  nil!!
(every!! '<!! (!! #(1 2 3)) (!! #(2 3 4 1)))  <=>  t!!
```

NOTES

Compiler Note:

The *Lisp compiler does not compile this operation.

REFERENCES

See the related functions **notany!!**, **notevery!!**, and **some!!**.

See also the general mapping function **amap!!**.

exp!! *[Function]*

Computes in parallel the value of *e* raised to the power specified by the supplied pvar.

SYNTAX

exp!! *numeric–pvar*

ARGUMENTS

numeric–pvar	Numeric pvar. Power to which *e* is raised.

RETURNED VALUE

exp–pvar	Temporary numeric pvar. Contains in each active processor the value of *e* raised to the power specified by the corresponding value of *numeric–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function computes and returns the value of *e* raised to the power *numeric–pvar* in each processor, where *e* is the base of the natural logarithms. Both complex and non-complex arguments are accepted.

EXAMPLES

```
(exp!! (!! 1))  <=>  (!! 2.7182817)
(exp!! (!! 3))          <=>    (!! 20.085535)
(exp!! (!! #c(2 2)))    <=>    (!! #c(-3.0749323 6.7188506))
```

---

expt!! [*Function*]

Computes in parallel the result of raising the first supplied pvar to the power specified by the second.

SYNTAX

expt!! *base–pvar power–pvar*

ARGUMENTS

base–pvar	Numeric pvar. Value to be raised to a power.
power–pvar	Numeric pvar. Power to which *base–pvar* is raised.

RETURNED VALUE

expt–pvar	Temporary numeric pvar. In each active processor, contains the result of raising the value of *base–pvar* to the power specified by the corresponding value of *power–pvar.*

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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

EXAMPLES

```
(expt!! (!! 2) (!! 3))  <=>  (!! 8)
```

NOTES

The function **expt!!** will signal an error if its arguments are of one pvar type, yet contain values that would produce a result of another pvar type.

For example, it is an error if *base–pvar* and *power–pvar* are integer pvars and *power–pvar* contains negative values in any processor. (This would produce a floating-point result for that processor.) Likewise, it is an error if *base–pvar* and *power–pvar* are floating-point pvars and *base–pvar* contains negative values in any processor. (This would produce a complex result in that processor.)

The reason **expt!!** is defined in this way is so that the pvar it returns can be guaranteed to be of a specific pvar type. If **expt!!** were allowed to return different data types in different processors, then it would have to return a general pvar as its result. Not only is this inefficient, it would also prevent **expt!!** expressions from compiling, because the *Lisp compiler does not compile expressions involving general pvars.

The general rule is that the **expt!!** function will not return a floating-point pvar as its result unless at least one of its arguments is already a floating-point pvar or has been coerced to a floating-point pvar by use of either **float!!** or **coerce!!**. Likewise, **expt!!** will not return a complex pvar as its result unless at least one of its arguments is already a complex pvar or has been coerced to a complex pvar by use of **complex!!** or **coerce!!**:

For example:

```
(expt!! 2 -3) ;; Inverse of cube of 2, signals error

(expt!! (!! 2) (float!! (!! -3))) <=>
(expt!! (!! 2) (coerce!! (!! -3) 'single-float-pvar)) <=>
(!! 0.125)
```

For example,

```
(expt!! -1 .5) ;; Square root of -1, signals an error

(expt!! (complex!! (!! -1)) (!! .5)) <=>
(expt!! (coerce!! (!! -1) 'single-complex-pvar) (!! .5)) <=>
(!! #c(-9.362676e-8 1.0))
```

As a side note, it is also an error for both base-pvar and power-pvar to be 0 in the same processor, unless power-pvar contains integer values — in this case, the result is the 1 coerced to the same data type as the value of base-pvar.

fceiling!! *[Function]*

Performs a parallel floating-point ceiling operation on the supplied pvar(s).

SYNTAX

fceiling!! *numeric–pvar* **&optional** *divisor–numeric–pvar*

ARGUMENTS

numeric–pvar — Non-complex numeric pvar. Value for which the floating-point ceiling is calculated.

divisor–numeric–pvar — Non-complex numeric pvar. If supplied, *numeric–pvar* is divided by *divisor–numeric–pvar* before the ceiling is taken.

RETURNED VALUE

fceiling–pvar — Temporary floating-point pvar. In each active processor, contains the floating-point ceiling of *numeric–pvar*, divided by *divisor–numeric–pvar* if supplied.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function is the parallel equivalent of the Common Lisp function **fceiling**. The value returned by **fceiling!!** is the same as that returned by **ceiling!!**, except that the result in each processor is always a floating-point number rather than an integer. The following forms are equivalent:

```
(fceiling!! data-pvar)  <=>  (float!! (ceiling!! data-pvar))
```

The argument pvars may contain either integer or floating-point values.

REFERENCES

See also these related rounding operations:

ceiling!!	**floor!!**	**round!!**	**truncate!!**

See also these related floating-point operations:

ffloor!!	**float!!**	
float-sign!!	**fround!!**	**ftruncate!!**
scale-float!!		

ffloor!! [*Function*]

Performs a parallel floating-point floor operation on the supplied pvar(s).

SYNTAX

ffloor!! *numeric–pvar* **&optional** *divisor–numeric–pvar*

ARGUMENTS

numeric–pvar Non-complex numeric pvar. Value for which the floating-point floor is calculated.

divisor–numeric–pvar
Non-complex numeric pvar. If supplied, *numeric–pvar* is divided by *divisor–numeric–pvar* before the floor is taken.

RETURNED VALUE

ffloor–pvar Temporary floating-point pvar. In each active processor, contains the floating-point floor of *numeric–pvar*, divided by *divisor–numeric–pvar* if supplied.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function is the parallel equivalent of the Common Lisp function **ffloor**. The value returned by **ffloor!!** is the same as that returned by **floor!!**, except that the result in each processor is always a floating-point number rather than an integer. The following forms are equivalent:

```
(ffloor!! data-pvar)  <=>  (float!! (floor!! data-pvar))
```

The argument pvars may contain either integer or floating-point values.

REFERENCES

See also these related rounding operations:

ceiling!!	**floor!!**	**round!!**	**truncate!!**

See also these related floating-point operations:

fceiling!!	**float!!**	
float–sign!!	**fround!!**	**ftruncate!!**
scale–float!!		

*fill

*[*Defun]*

Destructively modifies some or all elements in each sequence of the supplied sequence pvar to contain specified values.

SYNTAX

***fill** *sequence–pvar item–pvar* **&key :start :end**

ARGUMENTS

sequence–pvar	Sequence pvar. Pvar containing sequences to be modified.
item–pvar	Pvar containing values to be stored in sequences.
:start	Integer pvar. Zero-based index of sequence element at which to start fill operation. Default is **(!! 0)**.
:end	Integer pvar. Zero-based index of sequence element at which to end fill operation. Default is **(length!!** *sequence–pvar***)**.

RETURNED VALUE

sequence–pvar	Returns the modified sequence pvar.

SIDE EFFECTS

None.

DESCRIPTION

This function destructively modifies *sequence–pvar* by filling each sequence element with the value from *item–pvar*.

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 **:start** and **:end** arguments define a subsequence of elements to be modified in each sequence.

NOTES

Compiler Note:

The *Lisp compiler does not compile this operation.

REFERENCES

See also these related *Lisp sequence operators:

copy-seq!!	**length!!**	
***nreverse**	**reduce!!**	**reverse!!**
subseq!!		

See also the generalized array mapping functions **amap!!** and ***map**.

find!!, find-if!!, find-if-not!! [*Function*]

Perform a parallel search on a sequence pvar, returning in each processor the first sequence element that matches a given item or passes/fails a test.

SYNTAX

find!! *item-pvar sequence-pvar* **&key** **:test :test-not :start :end**
:key :from-end :return-value-if-not-found

find-if!! *test sequence-pvar*
&key :start :end :key :from-end :return-value-if-not-found

find-if-not!! *test sequence-pvar*
&key :start :end :key :from-end :return-value-if-not-found

ARGUMENTS

item-pvar	Pvar expression. Item to match in *sequence-pvar*. Must be of the same type as the elements of *sequence-pvar*.
test	One-argument pvar test. Used to test elements of *sequence-pvar*.
sequence-pvar	Sequence pvar. Contains sequences to be searched.
:test	Two-argument pvar predicate. Test used in comparisons. Indicates a match by returning a non-**nil** result. Defaults to **eql!!**.
:test-not	Two-argument pvar predicate. Test used in comparisons. Indicates a match by returning a **nil** result.
:start	Integer pvar. Index, zero-based, of sequence element at which search starts in each processor. If not specified, search begins with first element.
:end	Integer pvar. Index, zero-based, of sequence element at which search ends in each processor. If not specified, search continues to end of sequence.
:key	One-argument pvar accessor function. Applied to each element in *sequence-pvar* before test is performed.
:from-end	Boolean. Whether to begin search from end of sequence in all processors. Defaults to **nil**.

:return–value–if–not–found

Pvar expression. Value to return in processors where *sequence–pvar* does not contain the item in *item–pvar*. Default is **nil!!**.

RETURNED VALUE

find–pvar Temporary pvar, of same data type as elements of *sequence–pvar*. In each active processor, contains a copy of the first matching element of *sequence–pvar*. Contains the value of the argument **:return–value–if–not–found** for processors where no match is found.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

These functions are the parallel equivalent of the Common Lisp **find**, **find–if**, and **find–if–not** functions, with an additional keyword, **:return–value–if–not–found.**

In each processor, the function **find!!** searches *sequence–pvar* for elements that match *item–pvar*. It returns a pvar containing a copy of the first matching element found in each processor. Elements of *sequence–pvar* are tested against *item–pvar* with the **eql!!** operator unless another comparison operator is supplied as either of the **:test** or **:test–not** arguments. The keywords **:test** and **:test–not** may not be used together.

In each processor, the function **find–if!!** searches *sequence–pvar* for elements satisfying *test*. It returns a pvar containing a copy of the first matching element found in each processor. The function **find–if–not!!** searches *sequence–pvar* for elements failing *test*. It returns a pvar containing a copy of the first matching element found in each processor.

Arguments to the keywords **:start** and **:end** define a subsequence to be operated on in each processor.

The **:key** keyword accepts a user-defined function used to extract a search key from *sequence–pvar*. This key function must take one argument: an element of *sequence–pvar*.

In any processor failing the search, the value of the **:return-value-if-not-found** argument is returned.The keyword argument to **:return-value-if-not-found** must be a pvar and defaults to **nil!!**.

The keyword **:from-end** takes a boolean pvar that specifies from which end of *sequence-pvar* in each processor the operation will take place.

EXAMPLES

```
(find!! (!! 9) (!! #(1 4 9)))  <=>  (!! 9)
(find-if!! 'evenp!! (!! #(1 4 9)))  <=>  (!! 4)
(find-if-not!! 'evenp!! (!! #(1 4 9)))  <=>  (!! 1)
```

NOTES

Compiler Note:

The *Lisp compiler does not compile this operation.

REFERENCES

These functions are members of a group of similar sequence operators, listed below:

count!!	**count-if!!**	**count-if-not!!**
find!!	**find-if!!**	**find-if-not!!**
nsubstitute!!	**nsubstitute-if!!**	**nsubstitute-if-not!!**
position!!	**position-if!!**	**position-if-not!!**
substitute!!	**substitute-if!!**	**substitute-if-not!!**

See also the generalized array mapping functions **amap!!** and ***map**.

float!!

[Function]

Converts the numeric values of a specified pvar into a floating-point format.

SYNTAX

float!! *numeric–pvar* **&optional** *float–format–pvar*

ARGUMENTS

numeric–pvar	Non-complex numeric pvar. Pvar to be converted to floating-point format.
float–format–pvar	Floating-point pvar. If supplied, determines the floating-point format into which *numeric–pvar* is converted. Defaults to a pvar in single-float format.

RETURNED VALUE

float–pvar	Temporary numeric pvar. In each active processor, contains a copy of the value of *numeric–pvar* converted to floating-point format.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function converts any non-complex numeric pvar to a floating-point representation. In processors where *number–pvar* already contains floating-point numbers, those numbers are simply copied; elsewhere, **single–float** numbers are produced. When the optional argument *float–format–pvar* is given, *number–pvar* is converted to a matching floating-point format (single- or double-precision).

REFERENCES

See also these related floating-point operations:

fceiling!!	**ffloor!!**	
float-sign!!	**fround!!**	**ftruncate!!**
scale-float!!		

float–epsilon!! [*Function*]

Returns a pvar containing the smallest positive floating-point value representable in the format of the supplied floating-point pvar.

SYNTAX

float–epsilon!! *floating–point–pvar*

ARGUMENTS

floating–point–pvar
Floating-point pvar. Determines format of returned pvar.

RETURNED VALUE

epsilon–pvar — Temporary floating-point pvar. In each active processor, contains the smallest positive value representable in the same format as the corresponding value of *floating–point–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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.

REFERENCES

See also these related floating-point pvar limit functions:

least–negative–float!! **least–positive–float!!**
most–negative–float!! **most–positive–float!!** **negative–float–epsilon!!**

float–sign!! *[Function]*

Returns a unit value floating-point pvar with the same sign as the supplied pvar.

SYNTAX

float–sign!! *sign–pvar* **&optional** *value–pvar*

ARGUMENTS

sign–pvar	Floating-point pvar. Determines sign of result.
value–pvar	Floating-point pvar. Determines absolute value of result. Defaults to **(!! 1.0)**.

RETURNED VALUE

sign–value–pvar	Temporary floating-point pvar. In each active processor, contains a floating-point value with the same sign as *sign–pvar* and the same absolute value as *value–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function returns a floating-point pvar result with the same sign as *sign–pvar* and the same absolute value as *value–pvar*.

EXAMPLES

```
(float-sign!! (!! 0.08))     <=>   (!! 1.0)
(float-sign!! (!! -0.08))    <=>   (!! -1.0)
```

REFERENCES

See also these related floating-point operations:

fceiling!!	**ffloor!!**	**float!!**
fround!!	**ftruncate!!**	
scale–float!!		

floatp!! *[Function]*

Performs a parallel test for floating-point values on the supplied pvar.

SYNTAX

floatp!! *pvar*

ARGUMENTS

pvar	Numeric pvar. Pvar to be tested for floating-point values.

RETURNED VALUE

floatp-pvar	Temporary boolean pvar. Contains the value **t** in each active processor where the *pvar* contains a floating-point value, and **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This is the parallel equivalent of the Common Lisp function **floatp**. It returns the value **t** in each active processor where the *pvar* contains a floating-point value, and **nil** in all other active processors.

REFERENCES

See also these related pvar data type predicates:

booleanp!!	**characterp!!**	**complexp!!**
front–end–p!!	**integerp!!**	
numberp!!	**string–char–p!!**	**structurep!!**
typep!!		

floor!! *[Function]*

Performs a parallel floor operation on the supplied pvar(s).

SYNTAX

floor!! *numeric–pvar* **&optional** *divisor–numeric–pvar*

ARGUMENTS

numeric–pvar Non-complex numeric pvar. Value for which the floor is calculated.

divisor–numeric–pvar Non-complex numeric pvar. If supplied, *numeric–pvar* is divided by *divisor–numeric–pvar* before the floor is taken.

RETURNED VALUE

floor–pvar Temporary integer pvar. In each active processor, contains the floor of *numeric–pvar*, divided by *divisor–numeric–pvar* if supplied.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This is the parallel equivalent of the Common Lisp function **floor**, except that only one value — the floor of the quotient of *numeric–pvar* and *divisor–numeric–pvar* — is computed and returned.

REFERENCES

See also these related rounding operations:

ceiling!! **round!!** **truncate!!**

See also these related floating-point rounding operations:

fceiling!!	**ffloor!!**	**fround!!**	**ftruncate!!**

front–end!! [*Function*]

Returns a pvar whose values are references to a front-end object.

SYNTAX

front–end!! *scalar–object*

ARGUMENTS

scalar–object Front-end scalar object. Object referenced by the returned pvar.

RETURNED VALUE

front–end–pvar Temporary pvar. In each active processor, contains a reference to the front-end object *scalar–object.*

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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.

Front-end pvars may be passed as arguments only to *Lisp operations that access, move, or compare data, but not to operations that combine or compute with data. Operations that may take front-end pvar arguments include

eq!!	**if!!**	**news!!**
pref!!	**pref**	***pset**
scan!! (with **copy!!**)	***set**	**setf** (with **pref**)

EXAMPLES

Front-end pvars are useful for storing parallel data that has meaning when taken in combination with other data stored on the Connection Machine. For example, a front-end pvar can be used to store the symbolic names of a number of test subjects, such as simulated biological organisms. The expression

```
(*defvar names (front-end!! 'nothing))
```

defines a front-end pvar with symbolic values (initially, every value in **names** is a reference to the symbol **nothing**). Symbolic names can be stored into a front-end pvar by using **setf** with **pref**, as in

```
(setf (pref names 0) 'mutant-79)
```

Computations on other pvars can use the values stored in a front-end pvar for display or reference purposes, as in the examples below.

```
(*defun survivors ()
   (*when survived-simulated-catastrophe
      (format t "The survivors are:~%")
      (do-for-selected-processors (proc)
         (format t (pref names proc)))))

(*defun describe-microbe (bug-name)
   (*when (eq!! names (front-end!! bug-name))
      (format-description-for-selected-microbes)))
```

REFERENCES

See also the pvar allocation and deallocation operations

allocate!!	**array!!**	
***deallocate**	***deallocate–*defvars**	***defvar**
***let**	***let***	
make–array!!	**typed–vector!!**	**vector!!**
!!		

front–end–p!! [*Function*]

Performs a parallel test for front-end references on the supplied pvar.

SYNTAX

front–end–p!! *pvar*

ARGUMENTS

pvar Pvar expression. Tested in parallel for front-end reference values.

RETURNED VALUE

front–endp–pvar Temporary boolean pvar. Contains the value **t** in each active processor where the value of *pvar* is a front-end reference. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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.

EXAMPLES

```
(*defvar names (front-end!! 'nothing))

(front-end-p!! names) => t
```

REFERENCES

See also these related pvar data type predicates:

booleanp!!	**characterp!!**	**complexp!!**
floatp!!	**integerp!!**	
numberp!!	**string–char–p!!**	**structurep!!**
typep!!		

fround!! [*Function*]

Performs a parallel floating-point round operation on the supplied pvar(s).

SYNTAX

fround!! *numeric–pvar* **&optional** *divisor–numeric–pvar*

ARGUMENTS

numeric–pvar Non-complex numeric pvar. Value for which the floating-point round is calculated.

divisor–numeric–pvar
Non-complex numeric pvar. If supplied, *numeric–pvar* is divided by *divisor–numeric–pvar* before rounding is done.

RETURNED VALUE

fround–pvar Temporary floating-point pvar. In each active processor, contains the floating-point rounded value of *numeric–pvar*, divided by *divisor–numeric–pvar* if supplied.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function is the parallel equivalent of the Common Lisp function **fround**. The value returned by **fround!!** is the same as that returned by **round!!**, except that the result in each processor is always a floating-point number rather than an integer. The following forms are equivalent:

```
(fround!! data-pvar)  <=>  (float!! (round!! data-pvar))
```

The argument pvars may contain either integer or floating-point values.

REFERENCES

See also these related rounding operations:

ceiling!!	**floor!!**	**round!!**	**truncate!!**

See also these related floating-point operations:

fceiling!!	**ffloor!!**	**float!!**
float–sign!!	**ftruncate!!**	
scale–float!!		

ftruncate!! *[Function]*

Performs a parallel floating-point truncation on the supplied pvar(s).

SYNTAX

ftruncate!! *numeric–pvar* **&optional** *divisor–numeric–pvar*

ARGUMENTS

numeric–pvar Non-complex numeric pvar. Value for which the floating-point truncation is calculated.

divisor–numeric–pvar
Non-complex numeric pvar. If supplied, *numeric–pvar* is divided by *divisor–numeric–pvar* before truncation is done.

RETURNED VALUE

ftruncate–pvar Temporary floating-point pvar. In each active processor, contains the floating-point truncated value of *numeric–pvar*, divided by *divisor–numeric–pvar* if supplied.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function is the parallel equivalent of the Common Lisp function **ftruncate**. The value returned by **ftruncate!!** is the same as that returned by **truncate!!**, except that the result in each processor is always a floating-point number rather than an integer. The following forms are equivalent:

```
(ftruncate!! data-pvar)  <=>  (float!! (truncate!! data-pvar))
```

The argument pvars may contain either integer or floating-point values.

REFERENCES

See also these related rounding operations:

ceiling!!	**floor!!**	**round!!**	**truncate!!**

See also these related floating-point operations:

fceiling!!	**ffloor!!**	**float!!**
float-sign!!	**fround!!**	**scale-float!!**

*funcall [Macro]

Applies a parallel function defined by ***defun** to a set of arguments.

SYNTAX

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

ARGUMENTS

function	Symbol or function object. Function to call.
arguments	Scalar or pvar expressions. Arguments to pass to *function*.

RETURNED VALUE

returned–value	Scalar or pvar value. Value returned by *function*.

SIDE EFFECTS

None other than those of the supplied *function*.

DESCRIPTION

This is used just as Common Lisp's **funcall**, but is intended to be used with functions defined by ***defun**.

EXAMPLES

```
(*defun difference!! (pvar1 pvar2) (-!! pvar1 pvar2))
(*funcall 'difference!! (!! 3) (!! 4)))  <=>  (!! -1)
```

NOTES

Errors:

It is an error to use Common Lisp's **funcall** with a function defined using ***defun.** Also, just as **funcall** cannot be applied to macros, so ***funcall** cannot be applied to macros with the exception of operations defined by ***defun**. (Observant readers will notice that an operation defined by ***defun** actually *is* a macro in disguise — see the dictionary entry for ***defun** for more information.)

REFERENCES

See also the following related operations:

***apply**	***defun**	
***trace**	**un*defun**	***untrace**

gcd!! [*Function*]

Computes in parallel the greatest common denominator of the supplied integer pvars.

SYNTAX

gcd!! &rest *integer–pvars*

ARGUMENTS

integer–pvars	Integer pvars. Pvars for which gcd is to be calculated.

RETURNED VALUE

gcd–pvar	Temporary integer pvar. In each active processor, contains the greatest common denominator for the corresponding values of the *integer–pvars*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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*`)`

---

graphic–char–p!! [*Function*]

Performs a parallel test for graphic characters on the supplied pvar.

SYNTAX

graphic–char–p!! *character–pvar*

ARGUMENTS

character–pvar Character pvar. Tested in parallel for alphabetic characters. Must be a character pvar, a string-char pvar, or a general pvar containing only elements of type character or string-char.

RETURNED VALUE

graphic–charp–pvar

Temporary boolean pvar. Contains the value **t** in each active processor where the corresponding value of *character–pvar* is an graphic character. Contains **nil** in all other processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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.

gray–code–from–integer!! [*Function*]

Performs a parallel conversion from integers to Gray code values on the supplied pvar.

SYNTAX

gray–code–from–integer!! *integer–pvar*

ARGUMENTS

integer–pvar	Integer pvar. Pvar to be converted to gray code values. Must contain unsigned integers.

RETURNED VALUE

gray–code–pvar	Temporary unsigned integer pvar. In each active processor, contains the gray code representation of the corresponding value of *integer–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function converts each integer component of the *integer–pvar* argument into a Gray code representation. Binary reflected Gray code is used.

REFERENCES

See also the related function **integer–from–gray–code!!**.

grid [Function]

Creates and returns an address object containing the supplied integers as grid (NEWS) coordinates.

SYNTAX

grid &rest *integers*

ARGUMENTS

integers Scalar integers. Coordinates for the returned address object.

RETURNED VALUE

address–object Address object, allocated on front end. Contains the supplied *integers* as grid (NEWS) coordinates.

SIDE EFFECTS

None.

DESCRIPTION

This function creates and returns a front-end address object that contains the specified *integers* as grid (NEWS) coordinates.

EXAMPLES

```
(*cold-boot :initial-dimensions '(8 4))

(pref (self-address!!) (grid!! 4 2))  <=>  18
```

REFERENCES

See also the related operations

address-nth	**address-nth!!**
address-plus	**address-plus!!**
address-plus-nth	**address-plus-nth!!**
address-rank	**address-rank!!**
grid!!	
grid-relative!!	**self!!**

grid!! *[Function]*

Creates and returns an address-object pvar with grid (NEWS) coordinates specified by the supplied pvars.

SYNTAX

grid!! &rest *integer–pvars*

ARGUMENTS

integer–pvars	Integer pvars. Coordinates for the returned address-object pvar.

RETURNED VALUE

address–object–pvar

Temporary address-object pvar. In each active processor, contains an address object with the coordinates specified by the corresponding values of the *integer–pvars*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function creates and returns a pvar of address objects containing the specified *integer–pvars* as grid (NEWS) coordinates.

EXAMPLES

```
(*cold-boot :initial-dimensions '(8 4))

(pref!! (self-address!!) (grid!! (!! 4) (!! 2)))  <=>  (!! 18)
```

REFERENCES

See also the related operations

address-nth	**address-nth!!**
address-plus	**address-plus!!**
address-plus-nth	**address-plus-nth!!**
address-rank	**address-rank!!**
grid	
grid-relative!!	**self!!**

grid–from–cube–address [*Function*]

Converts a send (cube) address into a grid (NEWS) coordinate in the current VP set for a specified dimension.

SYNTAX

grid–from–cube–address *send–address dimension*

ARGUMENTS

send–address	Integer. Send address to be converted.
dimension	Integer. Number of the dimension for which the coordinate corresponding to *send–address* is to be returned. Zero-based.

RETURNED VALUE

coordinate	Integer. Grid (NEWS) coordinate in the current VP set, of the processor specified by *send–address* along the specified *dimension.*

SIDE EFFECTS

None.

DESCRIPTION

This function takes a *send–address* and returns the grid (NEWS) coordinate for the specified *dimension* in the current VP set. This function executes entirely in the front-end computer.

The *send–address* argument is a single integer representing the send address of a single processor. It is translated into a single integer representing the grid address of that processor along the specified *dimension.*

The *send–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.

EXAMPLES

Assume a four-dimensional machine configuration has been defined, and 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 for a given *send–address*, call **grid–from–cube–address** repeatedly, specifying a different *dimension* each time.

NOTES

Note that the send (cube) address corresponding to a particular grid address is not predictable from the grid address values alone. It also depends on the geometry of the current VP set, on the number of physical processors attached, and on the system software version in use. In particular, the relationship between send and grid addresses in the *Lisp simulator is different from that of the actual CM-2 hardware.

It is an error to rely on a specific, fixed relation between send and grid addresses except as provided by *Lisp address conversion functions such as **cube–from–grid–address**, **cube–from–vp–grid–address**, **grid–from–cube–address**, and **grid–from–vp–cube–address**.

REFERENCES

See also these related send and grid address translation operators:

cube–from–grid–address	**cube–from–grid–address!!**
cube–from–vp–grid–address	**cube–from–vp–grid–address!!**
grid–from–cube–address!!	
grid–from–vp–cube–address	**grid–from–vp–cube–address!!**
self–address!!	**self–address–grid!!**

grid–from–cube–address!! [*Function*]

Performs a parallel conversion from send (cube) addresses into grid (NEWS) coordinates in the current VP set.

SYNTAX

grid–from–cube–address!! *send–address–pvar dimension–pvar*

ARGUMENTS

send–address–pvar Integer pvar. Send address to be translated.

dimension–pvar Integer pvar. Number of the dimension for which the coordinate corresponding to *send–address–pvar* is to be returned. Zero-based.

RETURNED VALUE

coordinate–pvar Temporary integer pvar. In each processor, contains the grid (NEWS) coordinate in the current VP set, of the processor specified by *send–address–pvar* along the dimension specified by *dimension–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function takes a *send–address–pvar* and returns a pvar containing the grid (NEWS) coordinate in the current VP set for the specified *dimension–pvar* for each selected processor.

In each processor, this function translates the send (cube) address specified that processor's value of *send–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 *send-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 *send-address-pvar*.

EXAMPLES

Assume a four-dimensional machine configuration has been defined, and 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 in all active processors.

A more extensive example of **grid-from-cube-address!!** is detailed below.

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

(ppp (self-address!!) :mode :grid :end '(4 4) :format "~3D ")

  0   1   2   3
  8   9  10  11
 16  17  18  19
 24  25  26  27

(ppp (grid-from-cube-address!! (self-address!!) (!! 0))
       :mode :grid :end '(4 4) :format "~3D ")

  0   1   2   3
  0   1   2   3
  0   1   2   3
  0   1   2   3

(ppp (grid-from-cube-address!! (self-address!!) (!! 1))
       :mode :grid :end '(4 4) :format "~3D ")

  0   0   0   0
  1   1   1   1
  2   2   2   2
  3   3   3   3
```

NOTES

Note that the send (cube) address corresponding to a particular grid (NEWS) address is not predictable from the grid (NEWS) address values alone. It also depends on the geometry of the current VP set, on the number of physical processors attached, and on the system software version in use.

It is an error to rely on a specific, fixed relation between send and grid addresses except as provided by *Lisp address conversion functions such as **cube–from–grid–address!!**, **cube–from–vp–grid–address!!**, **grid–from–cube–address!!**, and **grid–from–vp–cube–address!!**.

REFERENCES

See also these related send and grid address translation operators:

cube–from–grid–address	**cube–from–grid–address!!**
cube–from–vp–grid–address	**cube–from–vp–grid–address!!**
grid–from–cube–address	
grid–from–vp–cube–address	**grid–from–vp–cube–address!!**
self–address!!	**self–address–grid!!**

grid–from–vp–cube–address [*Function*]

Converts a send (cube) address into a grid (NEWS) coordinate for the specified VP set.

SYNTAX

grid–from–vp–cube–address *vp–set send–address dimension*

ARGUMENTS

vp–set	VP set object. VP set in which the supplied *send–address* is converted.
send–address	Integer. Send address to be converted.
dimension	Integer. Number of the dimension for which the coordinate corresponding to *send–address* is to be returned. Zero-based.

RETURNED VALUE

coordinate	Integer. Grid (NEWS) coordinate in the specified *vp–set*, of the processor specified by *send–address* along the specified *dimension.*

SIDE EFFECTS

None.

DESCRIPTION

This function translates *send–address*, an integer representing the send address of a single processor in *vp–set*, into an integer representing the grid address of that processor along the specified *dimension* in *vp–set.*

The *send–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.

EXAMPLES

Assume that **my–vp** has a four-dimensional geometry, and assume that the processor referenced by send address 6534 has a grid address of (6 52 75 259) within the geometry of **my–vp**.

```
(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 for a given *send–address* in a given *vp–set*, call **grid–from–vp–cube–address** repeatedly, specifying a different *dimension* each time.

NOTES

Note that the send (cube) address corresponding to a particular grid (NEWS) address is not predictable from the grid (NEWS) address values alone. It also depends on the geometry of the current VP set, on the number of physical processors attached, and on the system software version in use.

It is an error to rely on a specific, fixed relation between send and grid addresses except as provided by *Lisp address conversion functions such as **cube–from–grid–address, cube–from–vp–grid–address**, **grid–from–cube–address**, and **grid–from–vp–cube–address.**

REFERENCES

See also these related send and grid address translation operators:

cube–from–grid–address	**cube–from–grid–address!!**
cube–from–vp–grid–address	**cube–from–vp–grid–address!!**
grid–from–cube–address	**grid–from–cube–address!!**
grid–from–vp–cube–address!!	
self–address!!	**self–address–grid!!**

grid–from–vp–cube–address!! [*Function*]

Performs a parallel conversion of send (cube) addresses into grid (NEWS) coordinates for the specified VP set.

SYNTAX

grid–from–vp–cube–address!! *vp–set send–address–pvar dimension–pvar*

ARGUMENTS

vp–set	VP set object. VP set for which grid (NEWS) coordinates are returned.
send–address–pvar	Integer pvar. Send address to be translated.
dimension–pvar	Integer pvar. Number of the dimension for which the coordinate corresponding to *send–address–pvar* is to be returned. Zero-based.

RETURNED VALUE

coordinate–pvar	Temporary integer pvar. In each processor, contains the grid (NEWS) coordinate in the specified *vp–set*, of the processor specified by *send–address–pvar* along the dimension specified by *dimension–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function performs a parallel conversion of send (cube) addresses into grid (NEWS) coordinates for the specified *vp–set*. This is the parallel equivalent of **grid–from–vp–cube–address**.

The value of *send–address–pvar* in each processor is assumed to be an integer representing the send address of a single processor in *vp–set.* This is translated into an integer representing the grid address of that processor along the dimension specified by the value of *dimension–pvar.*

The *send–address–pvar* 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.

EXAMPLES

Assume the VP set **my–vp** has a four-dimensional machine geometry, and that the processor referenced by send address 6534 has a grid address of (6 52 75 259) in **my–vp**.

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

Here, the grid address component corresponding to dimension 2 in **my–vp** is returned in all active processors.

NOTES

Note that the send (cube) address corresponding to a particular grid (NEWS) address is not predictable from the grid (NEWS) address values alone. It also depends on the geometry of the current VP set, on the number of physical processors attached, and on the system software version in use.

It is an error to rely on a specific, fixed relation between send and grid addresses except as provided by *Lisp address conversion functions such as **cube–from–grid–address!!**, **cube–from–vp–grid–address!!**, **grid–from–cube–address!!**, and **grid–from–vp–cube–address!!**.

REFERENCES

See also these related send and grid address translation operators:

cube–from–grid–address	**cube–from–grid–address!!**
cube–from–vp–grid–address	**cube–from–vp–grid–address!!**
grid–from–cube–address	**grid–from–cube–address!!**
grid–from–vp–cube–address	
self–address!!	**self–address–grid!!**

grid–relative!! [*Function*]

Returns an address-object pvar containing, for each processor, the grid (NEWS) coordinates of the processor a specified distance away along each dimension of the geometry of the current VP set.

SYNTAX

grid–relative!! &rest *relative–coord–pvars*

ARGUMENTS

relative–coord–pvars

Integer pvars. Specify relative distance along each dimension of the current VP set.

RETURNED VALUE

address–object–pvar

Temporary address-object pvar. In each active processor, contains an address object with the absolute grid (NEWS) coordinates of the processor specified by *relative–coord–pvars*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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)))
         ...)
```

REFERENCES

See also the related operations

address-nth	**address-nth!!**
address-plus	**address-plus!!**
address-plus-nth	**address-plus-nth!!**
address-rank	**address-rank!!**
grid	**grid!!**
self!!	

help

[Function]

Prints out a brief description of a *Lisp symbol.

SYNTAX

help &optional *symbol*

ARGUMENTS

symbol	*Lisp symbol. Symbol about which to print description.

RETURNED VALUE

nil	Evaluated for side effect only.

SIDE EFFECTS

Prints a brief description of the supplied symbol, or, if no symbol is supplied, a message describing where to find information about *Lisp.

DESCRIPTION

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 function defined by ***defun**, or a variable, and whether the symbol is new as of Connection Machine System Software Version 5.0.

---

*if [Macro]

Evaluates *Lisp forms with the currently selected set bound according to the logical value of a pvar expression.

SYNTAX

***if** *test–pvar then–form* **&optional** *else–form*

ARGUMENTS

test–pvar	Pvar expression. Selects processors in which to evaluate *then–form* and *else–form*.
then–form	Pvar expression. Evaluated with the currently selected set restricted to those processors for which the value of *test–pvar* is not **nil**.
else–form	Pvar expression. If supplied, evaluated with the currently selected set restricted to those processors in which the value of *test–pvar* is **nil**.

RETURNED VALUE

nil	Evaluated for side effect only.

SIDE EFFECTS

Temporarily restricts the currently selected set during the evaluation of *then–form* and *else–form*.

DESCRIPTION

This operator is analogous to the Common Lisp conditional **if**, with two essential differences. Both *then–form* and *else–form* are evaluated — in mutually exclusive sets of processors. Also, unlike Common Lisp's **if**, the ***if** macro returns no values and is executed only for its side effects.

The *then–form* argument is evaluated with the currently selected set bound to those processors in which *test–pvar* evaluates to a non-**nil** value. The optional *else–form* argument is evaluated with the currently selected set bound to those processors in which *test–pvar* evaluates to a **nil** value.

EXAMPLES

```
(*defvar winners)
(*defvar losers)
(*if  (zerop!! (random!! (!! 100)))
      (*set winners (!! 1))
      (*set losers (!! 1)))
```

Important: Even if no processors are selected by *test–pvar*, both *then–form* and *else–form* are evaluated.

```
(setq a 5 b 7)
(*if nil!! (setq a 7) (setq b 5))
a => 7
b => 5
```

In many cases, the macros ***if** and **if!!** can be used interchangeably. For example, these two expressions are equivalent, although in this case the latter expression is preferred as being more concise:

```
(*if  (evenp!! data-pvar)
      (*set bit-pvar (!! 1))
      (*set bit-pvar (!! 0)))
<=>
(*set bit-pvar (if!! (evenp!! data-pvar) (!! 1) (!! 0)))
```

As with all processor selection operators, calls to ***if** may be nested. Each call to ***if** subselects from the currently selected set, whether the selected set is the entire set of processors attached, or a subset selected by an enclosing operator. For example,

```
(*defvar result (!! 0))
(*if   (evenp!! (self-address!!))
       (*if  (zerop!! (mod!! (self-address!!) (!! 4)))
             (*set result (!! 4))
             (*set result (!! 2)))
       (*set result (!! 1)))
(ppp result)  =>  4 1 2 1 4 1 2 1 4 1 . . .
```

NOTES

Usage Note:

Forms such as **throw, return, return–from,** and **go** may be used to exit an external block or looping construct from within a processor selection operator. However, doing so will leave the currently selected set in the state it was in at the time the non-local exit form is executed. To avoid this, use the *Lisp macro **with–css–saved**. For example,

```
(block division
   (with-css-saved
      (*if (>!! y (!! 0))
        (if (*or (=!! (!! 0) x))
            (return-from division nil)
            (/!! y x)))))
```

Here **return–from** is used to exit from the **division** block if the value of **x** in any processor is zero. When the **with–css–saved** macro is entered, it saves the state of the currently selected set. When the code enclosed within the **with–css–saved** exits for any reason, either normally or via a call to a non-local exit operator like **return–from**, the currently selected set is restored to its original state.

See the dictionary entry for **with–css–saved** for more information.

Style Note:

As with the Common Lisp **if** operator, if no *else–form* is present, it is stylistically better to use the ***when** operator. Additionally, if the *test–pvar* is of the form

```
(*if (not!! test) ...
```

it is preferable to use the ***unless** operator, as in

```
(*unless test ...
```

REFERENCES

The *Lisp operator **if!!** behaves exactly like ***if**, but returns a pvar based on the evaluation of its arguments. See the dictionary entry for **if!!** for more information.

See also the related operators

***all**	***case**	**case!!**	***cond**	**cond!!**
***ecase**	**ecase!!**	***unless**	***when**	**with–css–saved**

if!! *[Macro]*

Returns a pvar obtained by evaluating *Lisp forms with the currently selected set bound according to the logical value of a pvar expression.

SYNTAX

if!! *test–pvar then–form* **&optional** *else–form*

ARGUMENTS

test–pvar — Pvar expression. Selects processors in which to evaluate *then–form* and *else–form.*

then–form — Pvar expression. Evaluated with the currently selected set restricted to those processors for which the value of *test–pvar* is not **nil**.

else–form — Pvar expression. If supplied, evaluated with the currently selected set restricted to those processors in which the value of *test–pvar* is **nil**. Defaults to **nil!!**.

RETURNED VALUE

then–else–pvar — Temporary pvar. Contains the value of *then–form* in all active processors where *test–pvar* evaluates to a non-**nil** value. Contains the value of *else–form* in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This operator is analogous to the Common Lisp conditional **if**, with one essential difference. Both *then–form* and *else–form* are evaluated, in mutually exclusive sets of processors.

The *then–form* argument is evaluated with the currently selected set bound to those processors in which *test–pvar* evaluates to a non-**nil** value. The optional *else–form* argument is evaluated with the currently selected set bound to those processors in which *test–pvar* evaluates to a **nil** value.

The **if!!** macro returns a pvar that contains the value of *then–form* in all processors in which *test–pvar* is non-**nil**, and the value of *else–form* in all processors in which *test–pvar* is **nil**.

```
(if!! question-pvar yes-pvar no-pvar)  <=>

(*let (result)
   (*when question-pvar (*set result yes-pvar))
   (*unless question-pvar (*set result no-pvar))
   result)
```

EXAMPLES

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

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

Important: Even if no processors are selected by *test–pvar*, both *then–form* and *else–form* are evaluated. For example,

```
(setq a 5 b 7)
(if!! nil!!
   (progn (setq a 7) (!! 0))
   (progn (setq b 5) (!! 1)))  =>  (!! 1)
a => 7
b => 5
```

In many cases, the macros ***if** and **if!!** can be used interchangeably. For example, these two expressions are equivalent, although in this case the latter expression is preferred as being more concise:

```
(*if  (evenp!! data-pvar)
      (*set bit-pvar (!! 1))
      (*set bit-pvar (!! 0)))

(*set bit-pvar (if!! (evenp!! data-pvar) (!! 1) (!! 0)))
```

As with all processor selection operators, calls to **if!!** may be nested. Each call to **if!!** subselects from the currently selected set, whether the selected set is the entire set of processors attached, or a subset selected by an enclosing operator. For example,

```
(*defvar result (!! 0))

(*set result
   (if!! (evenp!! (self-address!!))
         (if!!(zerop!! (mod!! (self-address!!) (!! 4)))
              (!! 4)
              (!! 2))
   (!! 1))

(ppp result)  =>  4 1 2 1 4 1 2 1 4 1 . . .
```

REFERENCES

The *Lisp operator ***if** behaves exactly like **if!!**, but does not return a pvar. See the dictionary entry for ***if** for more information.

See also the related operators

***all**	
***case**	**case!!**
***cond**	**cond!!**
***ecase**	**ecase!!**
***unless**	***when**
with-css-saved	

imagpart!! [*Function*]

Extracts the imaginary component from a complex pvar.

SYNTAX

imagpart!! *numeric–pvar*

ARGUMENTS

numeric–pvar Numeric pvar. Pvar from which imaginary component is extracted.

RETURNED VALUE

imagpart–pvar Temporary numeric pvar. In each active processor, contains the imaginary component of the corresponding value of *numeric–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function returns a temporary pvar containing in each processor the imaginary component of the complex value in *numeric–pvar*. Note that *numeric–pvar* need not be explicitly a complex-valued pvar. Non-complex values are automatically coerced into complex values with a zero imaginary component. Note that you can apply ***setf** to an **imagpart!!** call to modify the imaginary component of a complex numeric pvar.

REFERENCES

See also these related complex pvar operators:

abs!!	**cis!!**	**complex!!**
conjugate!!	**phase!!**	**realpart!!**

---

*incf [Macro]

Destructively increments each value of the supplied pvar.

SYNTAX

***incf** *numeric–pvar* **&optional** *value–pvar*

ARGUMENTS

numeric–pvar	Pvar expression. Pvar to be incremented.
value–pvar	Numeric pvar. Amount to add to *numeric–pvar*. Defaults to **(!! 1)**.

RETURNED VALUE

nil	Evaluated for side effect.

SIDE EFFECTS

Destructively increments each value of *pvar* by the corresponding value of *value–pvar*.

DESCRIPTION

Increments each element of *pvar* by the corresponding value of *value–pvar*. The *value–pvar* argument defaults to **(!! 1)**.

EXAMPLES

```
(*incf count-pvar (!! 3))
```

NOTES

Usage Note:

A call to the ***incf** macro expands as follows:

```
(*incf data-pvar (!! 4))
    ==>
(*setf data-pvar (+!! data-pvar (!! 4)))
```

For this reason, the *numeric–pvar* must be a modifiable pvar, such as a permanent, global, or local pvar. It is an error to supply a temporary pvar as the *numeric–pvar* to ***incf**.

REFERENCES

See also the related macro ***decf**.

The function **1+!!** can be used to non-destructively perform a addition by 1 on its argument pvar. See the dictionary entry on **1+!!** for more information.

initialize–character *[Function]*

Sets bit widths of *Lisp character attributes. If used, must be called prior to calling ***cold–boot**.

SYNTAX

initialize–character &key :code :bits :font :front–end–p :constantp

---

ARGUMENTS

:code	Integer. Number of bits to allocate for code attribute.
:bits	Integer. Number of bits to allocate for bits attribute.
:font	Integer. Number of bits to allocate for font attribute.
:front–end–p	Boolean value. Whether to directly copy character attribute widths used on the front end.
:constantp	Boolean value. Asserts whether or not the supplied values will remain constant for every succeeding call to ***cold–boot**. Used for optomization purposes by the *Lisp compiler.

RETURNED VALUE

nil	Evaluated for side effect only.

SIDE EFFECTS

Sets the values of the following global variables:

- ***char–bits–length**
- ***char–bits–limit**
- ***char–code–length**
- ***char–code–limit**
- ***char–font–length**

- ***char–font–limit**
- ***character–length**
- ***character–limit**

Determines whether the *Lisp compiler will assume that the bit widths of *Lisp character fields do not change.

DESCRIPTION

This function sets the values of the *Lisp character attributes, which are stored in global character variables. The **initialize–character** function must be called before ***cold–boot** is invoked, because these attributes are set when the machine is cold booted, not when the call to **initialize–character** is made.

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, defaulting to **nil**. It determines whether character pvar attribute widths should be copied from the format being used on the front end machine. If **:front–end–p** is **t**, the global character variables are set to match the character storage format of the front end machine.

The keyword **:constantp** takes a boolean value. This is used to assert whether or not the sizes of character attributes will remain constant across execution 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. Code compiled with **:constantp t** will run reliably only when the character attributes are the size specified at compile time. Code compiled with **:constantp nil** need not be recompiled to operate reliably with different character attribute sizes.

REFERENCES

For a discussion of Lisp character attributes, see the Characters chapter of *Common Lisp: The Language*.

See also the Connection Machine initialization function ***cold–boot**.

See also the initialization-list functions **add–initialization** and **delete–initialization.**

See also the related character pvar attribute operators:

char–bit!!	**char–bits!!**	**char–code!!**
char–font!!	**set–char–bit!!**	

int–char!! *[Function]*

Converts the supplied integer pvar into an character pvar.

SYNTAX

int–char!! *integer–pvar*

ARGUMENTS

integer–pvar	Integer pvar. Pvar to be converted.

RETURNED VALUE

character–pvar	Temporary character pvar. In each active processor, contains the character corresponding to the value of *integer–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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.

REFERENCES

See also the related character/integer pvar conversion operators:

char–code!!	**char–int!!**	**code–char!!**
digit–char!!	**int–char!!**	

integer-from-gray-code!! *[Function]*

Performs a parallel conversion from Gray code values to integers on the supplied pvar.

SYNTAX

integer-from-gray-code!! *gray-code-pvar*

ARGUMENTS

gray-code-pvar Integer pvar. Gray code value to be converted to a non-Gray-coded integer.

RETURNED VALUE

integer-pvar Temporary integer pvar. In each active processor, contains the integer value corresponding to the Gray code value in *integer-pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function treats each component of the argument pvar as a Gray-coded integer and converts it to a non-Gray-coded integer. The *gray-code-pvar* argument should contain unsigned integers.The function returns a pvar containing the unsigned results. Binary reflected Gray code is used.

REFERENCES

See also the related function **gray-code-from-integer!!**.

---

*integer–length [*Defun]

Determines the minimum bit-length needed to represent every value of an integer pvar.

SYNTAX

***integer–length** *integer–pvar*

ARGUMENTS

integer–pvar Integer pvar. Pvar for which minimum bit-length is determined.

RETURNED VALUE

integer–length Scalar integer. Minimum bit-length needed to represent every value of *integer–pvar*.

SIDE EFFECTS

None.

DESCRIPTION

This returns a scalar value that is the minimum bit-length needed to represent every integer value contined in *integer–pvar*. If no processors are selected, this function returns 0.

REFERENCES

See also the related global operators:

***and**	***logand**	
***logior**	***logxor**	***max**
***min**	***or**	***sum**
***xor**		

integer–length!! *[Function]*

Determines in parallel the minimum bit-length needed to represent each value of an integer pvar.

SYNTAX

integer–length!! *integer–pvar*

ARGUMENTS

integer–pvar	Integer pvar. Pvar for which minimum bit-lengths are determined.

RETURNED VALUE

length–pvar	Temporary integer pvar. In each active processor, contains the minimum bit-length needed to represent the corresponding value of *integer–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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.

EXAMPLES

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)
```

integer–reverse!! *[Function]*

Returns a pvar containing a bit-reversed copy of the values of the supplied integer pvar.

SYNTAX

integer–reverse!! *integer–pvar*

ARGUMENTS

integer–pvar	Integer pvar. Pvar containing values to be reversed.

RETURNED VALUE

reversed–pvar	Temporary integer pvar. In each active processor, contains a copy of the corresponding value of *integer–pvar* with the bits reversed, high-order exchanged with low-order and vice versa.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function returns an integer pvar of the same type and length as the argument. The result pvar contains a bit-reversed copy of *integer–pvar*'s bits, treated as an unsigned integer. The high-order bits become the low-order bits and vice versa.

NOTES

Usage Note:

This function relies on the internal representation of pvars in the Connection Machine system and therefore cannot work in the *Lisp simulator.

---

integerp!! *[Function]*

Performs a parallel test for integer values on the supplied pvar.

SYNTAX

integerp!! *pvar*

ARGUMENTS

pvar	Pvar expression. Pvar to be tested for integer values.

RETURNED VALUE

integerp–pvar	Temporary boolean pvar. Contains the value **t** in each processor where *pvar* contains an integer value. Contains the value **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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

REFERENCES

See also these related pvar data type predicates:

booleanp!!	**characterp!!**	**complexp!!**
floatp!!	**front–end–p!!**	
numberp!!	**string–char–p!!**	**structurep!!**
typep!!		

isqrt!! *[Function]*

Calculates in parallel the square root of the supplied integer pvar.

SYNTAX

isqrt!! *integer–pvar*

ARGUMENTS

integer–pvar	Integer pvar. Must contain only non-negative values. Pvar for which the square root is calculated.

RETURNED VALUE

isqrt–pvar	Integer pvar. In each active processor, contains the square root of the corresponding value of *integer–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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

---

lcm!! *[Function]*

Computes in parallel the least common multiple of the supplied integer pvars.

SYNTAX

lcm!! *integer–pvar* **&rest** *integer–pvars*

ARGUMENTS

integer–pvar, integer–pvars
Integer pvars. Pvars for which LCM is to be calculated.

RETURNED VALUE

lcm–pvar Temporary integer pvar. In each active processor, contains the least common multiple of the corresponding values of the *integer–pvars*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The function **lcm!!** takes one or more *integer–pvars* and computes, in each processor, the least common multiple of the values of the *integer–pvars* 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, the behavior is:

```
(lcm!! a  b) <=>
   (truncate!! (abs!! (*!! a b)) (gcd!! a b))
```

ldb!! [*Function*]

Extracts a byte in parallel from the supplied pvars.

SYNTAX

ldb!! *bytespec–pvar integer–pvar*

ARGUMENTS

bytespec–pvar	Byte specifier pvar, as returned from **byte!!**. Determines position and size of byte in *integer–pvar* that is extracted.
integer–pvar	Integer pvar. Integer from which byte is extracted.

RETURNED VALUE

byte–pvar	Temporary integer pvar. In each active processor, contains a copy of the byte of *integer–pvar* specified by *bytespec–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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)
```

REFERENCES

See also these related byte manipulation operators:

byte!!	**byte–position!!**	**byte–size!!**
deposit–byte!!	**deposit–field!!**	**dpb!!**
ldb–test!!	**load–byte!!**	**mask–field!!**

ldb–test!! *[Function]*

Tests in parallel whether a specified byte is non-zero in the supplied integer pvar.

SYNTAX

ldb–test!! *bytespec–pvar integer–pvar*

ARGUMENTS

bytespec–pvar	Byte specifier pvar, as returned from **byte!!**. Determines position and size of byte in *integer–pvar* which is tested.
integer–pvar	Integer pvar. Integer in which byte is tested.

RETURNED VALUE

byte–test–pvar	Temporary boolean pvar. Contains the value **t** in each active processor in which the byte of *integer–pvar* specified by *bytespec–pvar* is non-zero. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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**.

REFERENCES

See also these related byte manipulation operators:

byte!!	**byte–position!!**	**byte–size!!**
deposit–byte!!	**deposit–field!!**	**dpb!!**
ldb!!	**load–byte!!**	**mask–field!!**

least–negative–float!!
least–positive–float!! [*Function*]

Return a pvar containing the negative/positive floating-point value closest to zero in the format of the supplied floating-point pvar.

SYNTAX

least–negative–float!! *floating–point–pvar*
least–positive–float!! *floating–point–pvar*

ARGUMENTS

floating–point–pvar
Floating-point pvar. Determines format of returned pvar.

RETURNED VALUE

least–neg/pos–pvar Temporary floating-point pvar. In each active processor, contains the negative/positive floating-point value closest to zero and representable in the same format as the corresponding value of *floating–point–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function returns a floating-point pvar with the same format (single or double precision) as the argument *floating–point–pvar*. In each processor, the returned value is the negative (or positive) floating point number closest to zero in the floating-point format of *floating–point–pvar*.

EXAMPLES

The argument *floating–point–pvar* may be any floating point pvar of the required format. For example,

```
(least-negative-float!! (!! 0.0))  <=>  (!! -1.1754944e-38)

(least-positive-float!! (!! 0.0))  <=>  (!! 1.1754944e-38)
```

The same result would be obtained with any single-precision floating-point pvar argument.

REFERENCES

See also these related floating-point pvar limit functions:

float–epsilon!! **most–negative–float!!** **most–positive–float!!** **negative–float–epsilon!!**

length!! [Function]

Returns a pvar containing the lengths of the sequences in the supplied pvar.

SYNTAX

length!! *sequence–pvar*

ARGUMENTS

sequence–pvar Sequence pvar. Pvar containing sequences for which lengths are determined.

RETURNED VALUE

length–pvar Temporary integer pvar. Contains in each active processor the length of the sequence in *sequence–pvar.*

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function returns a positive integer pvar containing in each processor the number of elements in the corresponding sequence of *sequence–pvar.*

The argument *sequence–pvar* must be a vector pvar. The pvar returned by **length!!** holds the same value in each processor. The following forms are equivalent:

```
(length!! sequence-pvar)
<=>
(!! (*array-total-size sequence-pvar))
```

NOTES

Compiler Note:

The *Lisp compiler does not compile this operation.

REFERENCES

See also these related *Lisp sequence operators:

copy-seq!!	***fill**	
***nreverse**	**reduce!!**	**reverse!!**
subseq!!		

See also the generalized array mapping functions **amap!!** and ***map**.

*let, *let* *[Macro]*

Allocate local pvars that exist only during the evaluation of a set of forms.

SYNTAX

let(**&rest** *var–descriptors*)
  &optional *declarations*
  &body *body*

ARGUMENTS

var–descriptors A series of local pvar descriptors. Each descriptor can be either a list of the form (*symbol pvar–expression*) to specify a local pvar with an initial value, or just *symbol* to specify a local pvar without an initial value.

The *var–descriptor* components *symbol* and *pvar–expression* are described below.

symbol Symbol to which the corresponding local pvar is bound.

pvar–expression Pvar expression. Defines initial value of local pvar.

declarations Optional type declaration forms.

body *Lisp forms. Evaluated with the specified bindings in effect.

RETURNED VALUE

last–form–value Returns value of last *body* form evaluated. May be either a pvar or a front-end value. If a local pvar is returned, it becomes a temporary pvar.

SIDE EFFECTS

Allocates the specified local pvars on the stack during the evaluation of the *body* forms.

DESCRIPTION

The ***let** macro is used to allocate local pvars that exist only during the evaluation of a series of *Lisp forms. The ***let*** macro behaves identically to ***let** except that, as with Common Lisp's **let*** form, variable descriptors are evaluated in sequence, so that the value bound to each variable can be used in defining the values of succeeding variables.

The first argument of a call to ***let** must be a list containing any number of local pvar descriptors. Each descriptor can be a list consisting of a *symbol* that will name the local pvar, followed by a *pvar–expression* that will be used to initialize the pvar. Optionally, if no *pvar–expression* is required, the descriptor may be abbreviated to just the *symbol*.

The following call to ***let** illustrates the two possible *var–descriptor* forms:

```
(*let (no-init          ;;; this local pvar isn't initialized
       (inited (!! 0))) ;;; this pvar is initialized to 0
   (*set no-init inited)
   no-init)  =>  (!! 0)
```

The ***let** macro expects its first argument to be a list of pvar descriptors; even if no local pvars are defined, an empty list must be provided as the first argument to ***let**.

```
(*let ()
   (*!! (self-address!!) (!! 5)))
```

The *declarations* argument can be any number of *Lisp declaration forms. These forms can include, but are not limited to, type declarations for the local pvars defined by the variable descriptors of the ***let**.

Local pvars survive only for the extent of the supplied *body* forms, but may be accessed and modified by any functions these forms call. In other words, the *symbols* defined by the ***let** macro have lexical scope (as in Common Lisp), whereas the pvars themselves have dynamic extent that terminates when the ***let** form is exited.

The ***let** macro returns the value of the last form of *body*. If a local pvar is returned as the value of the ***let**, it becomes a temporary pvar and its contents should be copied into another pvar. The ***let** macro is *not* able to return multiple values.

EXAMPLES

This ***let** example "rolls" a pair of dice in each processor and returns the maximum roll value obtained in all processors as a single front-end value.

```
(*let  ((die1 (1+!! (random!! (!! 6))))
        (die2 (1+!! (random!! (!! 6)))))
   (declare (type (field-pvar 8) die1 die2))
   (*max (+!! die1 die2)))
```

This ***let*** example does the same thing. Notice that the value of the local pvar **dice-roll** depends on the values of the previously defined local pvars **die1** and **die2**.

```
(*let*  ((die1 (1+!! (random!! (!! 6))))
         (die2 (1+!! (random!! (!! 6))))
         (dice-roll (+!! die1 die2)))
   (*max dice-roll))
```

Here is a call to ***let** that defines only one local pvar. Note that the first argument to this ***let** call is still a list of lists.

```
(*let ((local-pvar (!! 3)))
   (*!! local-pvar (!! 5)))
```

The ***let** macro expects its first argument to be a list of local pvar descriptors. This expression would not work, for example, if it was mistakenly written as

```
(*let (local-pvar (!! 3))     ;;; Error: Not a list of lists
   (*!! local-pvar (!! 5)))
```

The ***let** macro is also able to allocate local pvars without initial values. In the following example, the pvars **x** and **y** are not initialized by the ***let** operator.

```
(*let (x
       y
       (scratch-pvar (!! 0)))
   (declare (type string-char-pvar x y))
   (*set x (get-first-data-pvar))
   (*set y (get-second-data-pvar))
   (operate-on-pvars x y scratch-pvar))
```

The contents of uninitialized local pvars such are not defined until values have been stored into them by an operator such as ***set**, as in the above example. It is an error to attempt to reference the contents of an unintialized pvar before its values have been defined in this way. For example, the following expression is in error, and its returned value is not defined:

```
(*let (x)
  (declare (type single-float-pvar x))
  (pref x 0)) ;;; Error: value of x has not yet been defined
```

In general it is wise to declare the pvars allocated by ***let**. This allows the *Lisp compiler to compile expressions involving those pvars. Here is the die-rolling example with **die1** and **die2** declared:

```
(*let ((die1 (1+!! (random!! (!! 6))))
       (die2 (1+!! (random!! (!! 6)))))
  (declare (type (field-pvar 8) die1 die2))
  (*max (+!! die1 die2)))
```

The length of a local pvar allocated by ***let** may be determined at run time.
For example:

```
(*let ((processor-address (self-address!!)))
  (declare
      (type (field-pvar *current-send-address-length*)
              processor-address))
  ... )
```

This type of declaration insures that pvars are defined efficiently, with the exact bit-size that is required.

A more complex type declaration example is provided by the following definition:

```
(defun make-me-a-float (type)
  (let ((s (if (eq type :single) 23 52))
        (e (if (eq type :single) 8 11)))
    (*let ((my-float (!! 0.0)))
      (declare (type (pvar (defined-float s e)) my-float))
      my-float)))
```

This function returns a floating point pvar of either single or double precision, depending on the value of its **type** argument.

Array pvars can be allocated on the *Lisp stack by declaring them appropriately from within a ***let** or a ***let*** form. However, when allocating an array using ***let** or ***let***, it is wise to explicitly declare the type of the pvar because undeclared pvars that have held any other type of data cannot hold arrays.

Here are some examples of the creation of local array pvars:

```
(*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)))
```

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:

- A list of dimension values, (*x y z*), may be given, such that *x*, *y*, and *z* each evaluate to integers at run time.
- A variable may be named. Its value at run time must be a list of integers.

For example:

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

Here, 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.

Any array pvar declaration form expects a list of integers specifying array dimensions. Consider the following two function defintions:

```
(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))
```

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 evaluating **(array-dimensions input-scalar-array)**, outside of the **declare** form. The symbol **dims** is then supplied to the **declare** form, which, when executed, finds **dims** properly bound to a list of integers.

NOTES

The *pvar-expression* forms used to initialize local pvars are evaluated in the currently selected set in effect outside the ***let** form, even if operators such as ***all** or ***when** are used in the body of the ***let** form to change the currently selected set.

The ***let** form `(*let ((x nil)) ... )` will not perform scalar promotion on the **nil** initialization form, because supplying **nil** as an initialization form indicates that the pvar **x** should not be initialized. The proper way to create a local pvar with **nil** in every processor is: `(*let ((x nil!!)) ... )`

REFERENCES

See also the pvar allocation and deallocation operations

allocate!!	**array!!**	***deallocate**	***deallocate–*defvars**
***defvar**	**front–end!!**	***let***	**make–array!!**
typed–vector!!		**vector!!**	**!!**

See also the *Lisp predicate **allocated–pvar–p**.

let–vp–set [*Function*]

Creates a temporary VP set that exists only during the evaluation of a set of forms.

SYNTAX

let–vp–set (*vp–set–name vp–set–creation–form*) **&body** *body*

ARGUMENTS

vp–set–name Symbol to which the temporary VP is bound.

vp–set–creation–form
VP set expression. Defines temporary VP set.

body *Lisp forms. Evaluated with *vp–set–name* bound to the VP set.

RETURNED VALUE

last–form–value Returns value of last *body* form evaluated.

SIDE EFFECTS

Allocates the specified VP set during the evaluation of the *body* forms, then deallocates it, using **deallocate–vp–set**.

DESCRIPTION

This macro creates a temporary VP set that may be used only within the supplied *body* forms. The symbol *vp–set–name* is bound to the VP set object returned by *vp–set–creation–form*, which should be either a call to **create–vp–set** or a form that makes such a call. The *body* forms are then executed. Finally, **deallocate–vp–set** is called to deallocate *vp–set–name* and the form is exited.

The returned value of **let–vp–set** is the value of the last form in *body*.

EXAMPLES

```
(progn
  (let-vp-set (temp-cube (create-vp-set '(32 32 32)))
    (*with-vp-set temp-cube
      (*let ((thoughts (!! 5))
             (random (random!! (!! 10))))
        (declare (type (field-pvar 8) thoughts random))
        (*set thoughts (*!! 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**.

REFERENCES

See also the following VP set definition and deallocation operators:

def–vp–set	**create–vp–set**
deallocate–def–vp–sets	**deallocate–vp–set**

See also the following flexible VP set operators:

allocate–vp–set–processors	**allocate–processors–for–vp–set**
deallocate–vp–set–processors	**deallocate–processors–for–vp–set**
set–vp–set–geometry	**with–processors–allocated–for–vp–set**

These operations are used to select the current VP set:

set–vp–set	***with–vp–set**

See also the following VP set information operations:

dimension–size	**dimension–address–length**
describe–vp–set	**vp–set–deallocated–p**
vp–set–dimensions	**vp–set–rank**
vp–set–total–size	**vp–set–vp–ratio**

*light

*[*Defun]*

Sets the pattern displayed on the front panel LEDs.

SYNTAX

***light** *boolean–pvar*

ARGUMENTS

boolean–pvar	Boolean pvar. Determines pattern displayed on front panel LEDs.

RETURNED VALUE

nil	Evaluated for side effect only.

SIDE EFFECTS

Sets front panel LEDs based on the value of the supplied *boolean–pvar*.

DESCRIPTION

This function provides control of the patterns displayed on the front-panel LEDs.

Each LED is connected to sixteen processors with sequential send addresses. The ***light** function affects only those LEDs for which all sixteen processors are selected. Each LED is turned on if all of its corresponding sixteen processors contain the value **nil** in *boolean–pvar*, and turned off if any processor is non-**nil**. The state (lit/unlit) of the remaining (unselected) LEDs is unchanged.

NOTES

Usage Note:

Before using the ***light** function, it is necessary to call the Paris function **CM:set–system–leds–mode** with an argument of **nil** to disconnect the LEDs from their normal processor monitoring mode, in which each LED is turned on whenever any of the sixteen processors to which that LED is connected are active.

*lisp [*Function*]

Switches between **user** and ***lisp** packages.

SYNTAX

***lisp &optional** *select–*lisp*

ARGUMENTS

*select–*lisp*	Boolean value or the keyword **:toggle**. If supplied, determines which package is selected. If not, defaults to **:toggle**.

RETURNED VALUE

None.	Returns no values.

SIDE EFFECTS

Changes the value of ***package***.

DESCRIPTION

The function ***lisp** makes switching the current package from **user** to ***lisp** and back again easy. It should be called only at top level. The *select–*lisp* argument determines which package is selected. A value of **t** sets the current package to ***lisp**. A value of **nil** sets the current package to **user**. The keyword **:toggle**, the default, toggles between the **user** and ***lisp** packages.

EXAMPLES

Called with an argument of **:toggle**, the default, the function ***lisp** toggles the current package between the **user** and ***lisp** packages:

```
(in-package 'user)

(*lisp :toggle)
Default package is now *LISP.

(*lisp)  ;;; :toggle is the default
Default package is now USER.
```

An argument of **t** forces selection of the ***lisp** package, and an argument of **nil** forces selection of the **user** package:

```
(in-package 'user)

(*lisp t)
Default package is now *LISP.

(*lisp t)
Default package is now *LISP.

(*lisp nil)
Default package is now USER.

(*lisp nil)
Default package is now USER.
```

NOTES

Editorial Note:

The ***lisp** function was written by William R. Swanson, who also compiled and edited the **Lisp Dictionary*.

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

Returns list containing the send addresses of all active processors.

SYNTAX

list–of–active–processors

ARGUMENTS

Takes no arguments.

RETURNED VALUE

send–address–list List of integers. Send addresses of all active processors.

SIDE EFFECTS

None.

DESCRIPTION

This simply returns a list of the send addresses of all the currently selected processors. The order of this list is not specified. This function could be written as:

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

REFERENCES

See also the definition of **loap**, a predefined alias for **list–of–active–processors**, and the looping operator **do–for–selected–processors.**

See also the related processor selection operators

***all**	
***if**	**if!!**
***case**	**case!!**
***cond**	**cond!!**
***ecase**	**ecase!!**
***unless**	***when**
with-css-saved	

load–byte!! [*Function*]

Extracts a byte in parallel from the supplied integer pvar.

SYNTAX

load–byte!! *integer–pvar position–pvar size–pvar*

ARGUMENTS

integer–pvar	Integer pvar. Pvar from which byte is extracted.
position–pvar	Integer pvar. Bit position, zero-based, of byte of *integer–pvar* to extract.
size–pvar	Integer pvar. Bit size of byte to extract.

RETURNED VALUE

byte–pvar	Temporary integer pvar. In each active processor, contains the byte of *integer–pvar* specified by *position–pvar* and *size–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The function **load–byte!!** extracts a byte in parallel from the supplied *integer–pvar*.

In each processor, this function extracts a byte from the value of *integer–pvar*, of size in bits specified by *size–pvar* and starting at the position specified by *position–pvar* (position 0 corresponds to the least significant bit). The following forms are equivalent:

```
(load-byte!! integer-pvar position-pvar size-pvar)
<=>
(ldb!! (byte!! size-pvar position-pvar) integer-pvar)
```

EXAMPLES

In any processor in which zero bits are extracted, the resulting field contains zero. Out-of-range bits are treated as zero for positive integers, and one for negative integers. For example,

```
(load-byte!! (!! 1)  (!! 2) (!! 3))    <=>  (!! 0)
(load-byte!! (!! -1) (!! 2) (!! 3))    <=>  (!! 7)
```

NOTES

Usage Note:

This operation is especially fast when both *position–pvar* and *size–pvar* are constants, as in

```
(load-byte!! data-pvar (!! 2) (!! 3))
```

REFERENCES

See also these related byte manipulation operators:

byte!!	**byte–position!!**	**byte–size!!**
deposit–byte!!	**deposit–field!!**	**dpb!!**
ldb!!	**ldb–test!!**	**mask–field!!**

loap

[Macro]

Returns list containing the send addresses of all active processors.

SYNTAX

loap

ARGUMENTS

Takes no arguments.

RETURNED VALUE

address–list	List of integers, representing the send addresses of all of the active processors.

SIDE EFFECTS

None.

DESCRIPTION

This macro is an alias for **list–of–active–processors**.

---

*locally [*Macro*]

Provides the *Lisp compiler with declarations that remain in effect for the duration of a body form.

SYNTAX

***locally** *declaration–1 declaration–2 ... declaration–n* **&body** *body*

ARGUMENTS

declaration–n	Declaration forms.
body	*Lisp forms. Compiled with the specified declarations in effect.

RETURNED VALUE

last–form–value	Returns value of last *body* form evaluated.

SIDE EFFECTS

None.

DESCRIPTION

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.

The *Lisp compiler's code walker largely eliminates any need to use the ***locally** operator. See Chapter 4, "*Lisp Type Declaration," for a description of this feature and of other operators that should be used instead of ***locally**.

EXAMPLES

A simple example of the use of ***locally** is

```
(setq allocated-pvar
   (allocate!! (!! 0.0) nil 'single-float-pvar))

(*locally
   (declare (type single-float-pvar allocated-pvar))
   (*let (result-pvar)
      (*set allocated-pvar (random!! (!! 10.0))))
   (dotimes (i 3)
      (*incf result-pvar allocated-pvar)))
```

in which **allocated–pvar** is declared to be of type **single–float–pvar**.

An example of the use of ***locally** in a function definition is

```
(defun *locally-test (j)
  (*locally
    (declare (type fixnum j))
    (*let (temp)
      (declare (type (unsigned-byte-pvar 32) temp))
      (*set temp (!! j))
      (ppp temp :end 8))))
```

The use of ***locally** in this function declares the type of the scalar argument **j**, allowing this function to execute more efficiently in both interpreted and compiled form.

```
(*locally-test 1.0)
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
```

The following example displays many of the locations in which ***locally** can be used to provide a localized declaration.

```
(*cold-boot :initial-dimensions '(8 4))

(*proclaim '(type single-float-pvar result-pvar))
(*defvar result-pvar)

(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 *number-of-processors-limit*)
            (*locally
              (declare (type fixnum i))
              (*let ((temp (*!! (+!! (float!! (!! i)) (!! j))
                                 (sin!! (!! j)))))
                (declare (type single-float-pvar temp))
                (local-pvar-function temp)
                ))))))))

(*locally-example result-pvar)

(ppp result-pvar :end 6)
5.94665 5.94665 5.94665 5.94665 5.94665 5.94665
```

REFERENCES

See also the related *Lisp declaration operators:

***proclaim** **unproclaim**

See also the related type translation function **taken-as!!**.

See also the related type coercion function **coerce!!**.

log!! [*Function*]

Takes the logarithm of the supplied pvar.

SYNTAX

log!! *numeric–pvar* **&optional** *base–pvar*

ARGUMENTS

numeric–pvar	Numeric pvar. Pvar for which logarithm is calculated.
base–pvar	Numeric pvar. If supplied, determines base in which logarithm is calculated. Defaults to base of natural logarithms.

RETURNED VALUE

log–pvar	Numeric pvar. In each active processor, contains logarithm of the corresponding value of *numeric–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function returns the logarithm of the argument *numeric–pvar* in the base *base–pvar*. If *base–pvar* is absent, 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.

EXAMPLES

```
(log!! (!! 4) (!! 2))  <=>  (!! 2.0)
```

NOTES

The function **log!!** will never return a complex pvar as its result unless *numeric-pvar* is complex, or is coerced into complex form by use of the functions **complex!!** or **coerce!!**, as shown below.

```
(log!!   (coerce!! (!! -1) '(pvar (complex single-float))))
    <=>
(log!!   (complex!! (!! -1.0)))
    <=>
(!! #c(0.0 3.1415927))
```

*logand [*Defun]

Returns bitwise logical AND of all values in the supplied integer pvar.

SYNTAX

***logand** *integer–pvar*

ARGUMENTS

integer–pvar Integer pvar. Pvar for which logical AND is calculated.

RETURNED VALUE

logand–integer Integer. Bitwise logical AND of all values in *integer–pvar*.

SIDE EFFECTS

None.

DESCRIPTION

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

EXAMPLES

```
(*logand (!! 7))  =>  7
(*when nil!! (*logand (!! 7)))  =>  -1
(*logand (if!!(evenp!! (self-address!!))
              (!! 6)
              (!! 3)))  =>  2
(*logand (!! 0))  =>  0
```

REFERENCES

See also the related global operators:

***and**	***integer-length**	
***logior**	***logxor**	***max**
***min**	***or**	***sum**
***xor**		

See also the related logical operators:

and!!	**not!!**	**or!!**	**xor!!**

logand!!, logandc1!!, logandc2!!, logeqv!!, logior!!, lognand!!, lognor!!, lognot!!, logorc1!!, logorc2!!, logxor!! [*Function*]

Perform parallel bitwise logical operations on the supplied integer pvars.

SYNTAX

lognot!!	*integer–pvar*	
logand!!	**&rest**	*integer–pvars*
logeqv!!	**&rest**	*integer–pvars*
logior!!	**&rest**	*integer–pvars*
logxor!!	**&rest**	*integer–pvars*
logandc1!!	*integer–pvar1*	*integer–pvar2*
logandc2!!	*integer–pvar1*	*integer–pvar2*
lognand!!	*integer–pvar1*	*integer–pvar2*
lognor!!	*integer–pvar1*	*integer–pvar2*
logorc1!!	*integer–pvar1*	*integer–pvar2*
logorc2!!	*integer–pvar1*	*integer–pvar2*

ARGUMENTS

integer–pvar(s) Integer pvars. Combined using bitwise logical operations.

RETURNED VALUE

logand–pvar Temporary integer pvar. In each active processor, contains the bitwise logical combination of the supplied *integer–pvars*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

These functions perform logical bitwise operations on their arguments.

The logical operation performed by each *Lisp function is:

logand!!	Bitwise logical AND
logandc1!!	Bitwise logical AND, with *integer–pvar–1* complemented
logandc2!!	Bitwise logical AND, with *integer–pvar–2* complemented
logeqv!!	Bitwise logical equivalence
logior!!	Bitwise logical inclusive OR
lognand!!	Bitwise logical NAND
lognor!!	Bitwise logical NOR
lognot!!	Bitwise logical NOT
logorc1!!	Bitwise logical inclusive OR, with *integer–pvar–1* complemented
logorc2!!	Bitwise logical inclusive OR, with *integer–pvar–2* complemented
logxor!!	Bitwise logical exclusive OR

For functions that accept any number of *integer–pvar* arguments, the value returned if no pvars are provided is **(!! –1)** for **logand!!** and **logeqv!!**, **(!! 0)** for **logior!!** and **logxor!!**.

EXAMPLES

```
(logand!! (!! 7) (!! 7))          <=>  (!! 7)
(logand!! (!! 7) (!! 3))          <=>  (!! 3)
(logand!! (!! 7) (!! 6) (!! 3))  <=>  (!! 2)
(logand!! (!! 7) (!! 0))              <=>  (!! 0)

(logandc1!! pvar1 pvar2)
                    <=> (logand!! (lognot!! pvar1) pvar2)
(logandc2!! pvar1 pvar2)
                    <=> (logand!! pvar1 (lognot!! pvar2))

(logeqv!! (!! 7) (!! 7))          <=>  (!! -1)
(logeqv!! (!! 7) (!! 3))          <=>  (!! -5)
(logeqv!! (!! 7) (!! 6) (!! 3))  <=>  (!! 2)
(logeqv!! (!! 7) (!! 0))          <=>  (!! -8)

(logior!! (!! 0))                 <=>  (!! 0)
(logior!! (!! 7) (!! 7))          <=>  (!! 7)
(logior!! (!! 7) (!! 3))          <=>  (!! 7)
(logior!! (!! 4) (!! 1) (!! 0))  <=>  (!! 5)

(lognand!! pvar1 pvar2)
                    <=> (lognot!! (logand!! pvar1 pvar2))
(lognor!! pvar1 pvar2)
                    <=> (lognot!! (logior!! pvar1 pvar2))
(logorc1!! pvar1 pvar2)
                    <=> (logior!! (lognot!! pvar1) pvar2)
(logorc2!! pvar1 pvar2)
                    <=> (logior!! pvar1 (lognot!! pvar2))
```

```
(lognot!! (!! -1)) <=> (!! 0)

(logxor!! (!! 7) (!! 7))                   <=>  (!! 0)
(logxor!! (!! 1) (!! 3) (!! 4))            <=>  (!! 6)
(logxor!! (!! 0) (!! 1) (!! 2) (!! 4))     <=>  (!! 7)
```

NOTES

Usage Note

Like Common Lisp, *Lisp concentually represents integer pvars as having infinitely many bits, that is, *Lisp sign extends the 1 or a 0 sign bit of an integer pvar as many bits as needed. This means that performing **lognot!!** on a non-negative integer pvar will result in a signed integer pvar with negative values:

```
(*proclaim '(type (field-pvar 2) x))

(*defvar x 1)

(ppp (lognot!! x) :end 4)
 -2 -2 -2 -2
```

Attempting to perform

```
(*set x (lognot!! x))
```

will not work because **x** has been declared unsigned, and the call to **lognot!!** will return a signed integer pvar, which ***set** would then attempt to copy back into the unsigned integer pvar **x**.

To do an "unsigned" **lognot!!**, try something like this:

```
(*set x (load-byte!! (lognot!! x) 0 xlen))
```

where **xlen** is the original length of **x**, in bits.

logbitp!! *[Function]*

Tests in parallel whether a specified bit of the supplied integer pvar is set.

SYNTAX

logbitp!! *index–pvar integer–pvar*

ARGUMENTS

index–pvar	Integer pvar. Index, zero-based, of bit to be tested.
integer–pvar	Integer pvar. Pvar on which parallel bit test is performed.

RETURNED VALUE

logbitp–pvar	Temporary boolean pvar. Contains the value **t** in each active processor where the bit in *integer–pvar* specified by *index–pvar* is set (equal to 1). Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This predicate function tests in parallel whether a specified bit of the supplied integer pvar is set. In each processor, **logbitp!!** examines the bit specified by *index–pvar* in the value of *integer–pvar*, where 0 specifies the least significant bit. The returned pvar has the value **t** wherever the selected bit is a one-bit; otherwise it has the value **nil**.

```
(logbitp!! index-pvar byte-pvar)
    <=>
(plusp!! (ldb!! (byte!! index-pvar (!! 1)) byte-pvar))
```

logcount!! [*Function*]

Determines in parallel the number of set bits in an integer pvar.

SYNTAX

logcount!! *integer–pvar*

ARGUMENTS

integer–pvar Integer pvar. Pvar in which set bits are counted.

RETURNED VALUE

bitcount–pvar Integer pvar. In each active processor, contains the number of set bits in the corresponding value of *integer–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function determines, in each processor, the number of one-bits in that processor's value of *integer–pvar* and returns a non-negative integer pvar containing the result. 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.

EXAMPLES

```
(ppp (logcount!! (self-address!!)))  =>
0 1 1 2 1 2 2 3 1 2 2 3 2 3 3 4 . . .

(logcount!! (!! 7))  <=>  (!! 3)
```

---

*logior [*Defun]

Returns bitwise logical inclusive OR of all values in the supplied integer pvar.

SYNTAX

***logior** *integer–pvar*

ARGUMENTS

integer–pvar	Integer pvar. Pvar for which logical inclusive OR is calculated.

RETURNED VALUE

logior–integer	Integer. Bitwise logical inclusive OR of all values in *integer–pvar*.

SIDE EFFECTS

None.

DESCRIPTION

This returns a Lisp value that is the bitwise logical inclusive OR of the contents of *integer–pvar* in all selected processors. This returns the Lisp value 0 if there are no selected processors.

EXAMPLES

```
(*logior (!! 7))  =>  7
(*when nil!! (*logior (!! 7)))  =>  0
(*logior (if!!(evenp!! (self-address!!))
               (!! 6)
               (!! 3)))  =>  7
(*logior (!! 0))  =>  0
```

REFERENCES

See also the related global operators:

***and**	***integer–length**	***logand**
***logxor**	***max**	
***min**	***or**	***sum**
***xor**		

See also the related logical operators:

and!!	**not!!**	**or!!**	**xor!!**

logtest!! *[Function]*

Performs a parallel test on the supplied integer pvars for bits which are set in both pvars.

SYNTAX

logtest!! *integer-pvar1 integer-pvar2*

ARGUMENTS

integer-pvar1, integer-pvar2
Integer pvars. Tested in parallel for bits set in both pvars.

RETURNED VALUE

logtest-pvar Temporary boolean pvar. Contains the value **t** in each active processor where the values of *integer-pvar1* and *integer-pvar2* contain corresponding bits that are set in both pvars. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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!! pvar1 pvar2)
    <=>
(not!! (zerop!! (logand!! pvar1 pvar2)))
```

*logxor [*Defun]

Returns bitwise logical XOR of all values in the supplied integer pvar.

SYNTAX

***logxor &rest** *integer–pvar*

ARGUMENTS

integer–pvar Integer pvar. Pvar for which logical inclusive XOR is calculated.

RETURNED VALUE

logxor–integer Integer. Bitwise logical inclusive XOR of all values in *integer–pvar*.

SIDE EFFECTS

None.

DESCRIPTION

This returns a Lisp value that is the bitwise logical exclusive OR of the contents of *integer–pvar* in all selected processors. This returns the Lisp value 0 if there are no selected processors.

EXAMPLES

```
(*let ((test (!! 0)))
   (*setf (pref test 0) 1)
   (*setf (pref test 1) 2)
   (*setf (pref test 2) 4)
   (*logxor test))
=>  7
```

REFERENCES

See also the related global operators:

***and**	***integer-length**	***logand**
***logior**	***max**	
***min**	***or**	***sum**
***xor**		

See also the related logical operators:

and!!	**not!!**	**or!!**	**xor!!**

lower–case–p!! [*Function*]

Performs a parallel test for lowercase characters on the supplied pvar.

SYNTAX

lower–case–p!! *character–pvar*

ARGUMENTS

character–pvar Character pvar. Tested in parallel for lowercase characters.

RETURNED VALUE

lowercasep–pvar Temporary boolean pvar. Contains the value **t** in each active processor where the corresponding value of *character–pvar* is a lowercase alphabetic character. Contains **nil** in all other processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This predicate returns a pvar that has the value **t** in each processor where the supplied *character–pvar* contains a lowercase character, and the value **nil** in all other processors.

---

make–array!! [*Function*]

Creates and returns an array pvar.

SYNTAX

make–array!! *dimensions* **&key :element–type :initial–element**

ARGUMENTS

dimensions	Integer, or list of integers. Dimensions of array pvar.
:element–type	Common Lisp or *Lisp type specifier. Specifies data type of elements, and must be supplied.
:initial–element	Scalar or pvar value. If supplied, determines initial value of array elements.

RETURNED VALUE

array–pvar	Temporary array pvar with the specified *dimensions*. Data type and initial contents are as specified by the **:element–type** and **:initial–element** arguments.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The function **make–array!!** returns an array pvar on the *Lisp stack.

The *dimensions* argument is either a single non-negative integer or a list of non-negative integers. Each integer must be less than ***array–dimension–limit**. If a list of dimensions is given, the length of the list must be less than ***array–rank–limit**. The product of all dimensions must be smaller than ***array–total–size–limit**.

Any valid fixed-size Common Lisp type or pvar type of fixed size may be specified as the value of **:element–type**. It is an error to *not* provide an **:element–type** argument when calling **make–array!!**.

The value of **:initial–element** may be either a front-end scalar or a pvar. If it is a scalar, the function **!!** is used to convert it to a constant pvar. In either case, **make–array!!** stores the value of *initial–element* in each processor into each element of the corresponding array. If *initial–element* is not specified, the contents of the newly created array are undefined.

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**.

EXAMPLES

```
(*defvar new-array-pvar)
(*set new-array-pvar
   (make-array!! '(2 2 2)
      :element-type '(complex single-float)
      :initial-element #c(5.3 0.0)))

(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.

REFERENCES

See also the pvar allocation and deallocation operations

allocate!!	**array!!**	
***deallocate**	***deallocate–*defvars**	***defvar**
front–end!!	***let**	***let***
typed–vector!!	**vector!!**	**!!**

make–char!! [*Function*]

Creates and returns a copy of a character pvar with modified bits and font attributes.

SYNTAX

make–char!! *character–pvar* **&optional** *bits–pvar font–pvar*

ARGUMENTS

character–pvar	Character pvar. Determines code attribute of returned character pvar.
bits–pvar	Integer pvar. If supplied, determines bits attribute of returned character pvar. Defaults to **(!! 0)**.
font–pvar	Integer pvar. If supplied, determines font attribute of returned character pvar. Defaults to **(!! 0)**.

RETURNED VALUE

char–pvar	Character pvar. In each active processor, contains a copy of the corresponding value of *character–pvar*, with bits and font attributes as specified by *bits–pvar* and *font–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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

REFERENCES

See also the related character pvar constructor **character!!**.

See also the related character pvar attribute operators:

char–bit!!	**char–bits!!**	**char–code!!**
char–font!!	**initialize–character**	**set–char–bit!!**

*map [*Function*]

Maps a function in parallel over the supplied array pvars.

SYNTAX

***map** *operator* **&rest** *array–pvars*

ARGUMENTS

operator	Symbol or functional object. Function to be applied.
array–pvars	Array pvars. Pvars containing arrays that *function* is mapped over.

RETURNED VALUE

nil	Evaluated for side effect only.

SIDE EFFECTS

None other than those of the supplied *function*.

DESCRIPTION

The ***map** function maps the supplied *operator* over the supplied array pvars. The *operator* is applied in turn to each set of elements having the same row-major index in the supplied *array–pvars*. Thus, the *n*th time *function* is called, it is applied to a list containing the *n*th element in row-major order from each of the *array–pvars*.

The *Lisp function ***map** is similar to the Common Lisp function **map**, but while **map** works only on vectors, ***map** works on any type of array pvar.

For vectors, ***map** behaves much like **map** in accepting vector pvar arguments of different element sizes and in limiting the mapping operation to the length of the shortest vector pvar supplied. For all other types of array pvars, however, ***map** expects the array sizes of the supplied *array–pvars* to be identical.

EXAMPLES

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. This can be accomplished by

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

(*defvar matrix1
   (!! #2A((1.0 2.0 3.0) (4.0 5.0 6.0) (7.0 8.0 9.0))))
(*defvar matrix2
   (!! #2A((3.0 2.0 1.0) (6.0 5.0 4.0) (9.0 8.0 7.0))))

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

(*map-example 2.0)

(pref matrix1 0)
=> #2A((8.0 8.0 8.0) (20.0 20.0 20.0) (32.0 32.0 32.0))
```

REFERENCES

See also the related function **amap!!**.

mask–field!! *[Function]*

Copies a bit field in parallel from the supplied integer pvar.

SYNTAX

mask–field!! *bytespec–pvar integer–pvar*

ARGUMENTS

bytespec–pvar	Byte specifier pvar, as returned from **byte!!**. Determines position and size of bit field in *integer–pvar* which is copied.
integer–pvar	Integer pvar. Integer from which bit field is copied.

RETURNED VALUE

newbyte–pvar	Temporary integer pvar. In each active processor, contains an integer that agrees with the corresponding value of *integer–pvar* in the bit field specified by *bytespec–pvar*, and has zero bits elsewhere.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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 *newbyte–pvar* result therefore agrees with *integer–pvar* in the byte specified, but has zero bits everywhere else.

The following forms are equivalent:

```
(mask-field (byte!! size-pvar pos-pvar) bits-pvar)
    <=>
(logand!! bits-pvar
          (dbp!! (!! -1) (byte!! size-pvar pos-pvar) 0))
```

REFERENCES

See also these related byte manipulation operators:

byte!!	**byte–position!!**	**byte–size!!**
deposit–byte!!	**deposit–field!!**	**dpb!!**
ldb!!	**ldb–test!!**	**load–byte!!**

*max [*Defun]

Returns the maximum numeric value contained in the supplied pvar.

SYNTAX

***max** *numeric–pvar*

ARGUMENTS

numeric–pvar Numeric pvar. Pvar for which maximum value is determined.

RETURNED VALUE

max–value Scalar value. Maximum numeric value contained in the *numeric–pvar*.

SIDE EFFECTS

None.

DESCRIPTION

This returns a scalar value that is the maximum of the contents of *numeric–pvar* in all selected processors. This returns the Lisp value **nil** if there are no selected processors.

EXAMPLES

```
(*max (mod!! (self-address!!) (!! 5)))  <=>  4
```

REFERENCES

See also the related global operators:

***and**	***integer–length**	***logand**	***or**	***xor**
***logior**	***logxor**	***min**	***sum**	

---

max!! *[Function]*

Determines in parallel the maximum numeric value of the supplied pvars.

SYNTAX

max!! *numeric–pvar* **&rest** *numeric–pvars*

ARGUMENTS

numeric–pvar, numeric–pvars
Non-complex numeric pvars. Pvars for which the maximum value is determined.

RETURNED VALUE

max–pvar Temporary numeric pvar. In each active processor, contains the maximum of the corresponding values of the supplied *numeric–pvar* arguments.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This returns a pvar that contains in each processor the maximum of the corresponding values of the supplied *numeric–pvars* in that processor.

EXAMPLES

```
(ppp (max!!   (mod!! (self-address!!) (!! 2))
              (mod!! (self-address!!) (!! 3))))  =>
0 1 2 1 1 2 0 1 2 1 1 2 0 1 2 . . .
```

---

*min

[*Defun]

Returns the minimum numeric value contained in the supplied pvar.

SYNTAX

***min** *numeric–pvar*

ARGUMENTS

numeric–pvar	Numeric pvar. Pvar for which minimum value is determined.

RETURNED VALUE

min–value	Scalar value. Minimum numeric value contained in the *numeric–pvar.*

SIDE EFFECTS

None.

DESCRIPTION

This returns a scalar value that is the minimum of the contents of *numeric–pvar* in all selected processors. It returns the Lisp value **nil** if there are no selected processors.

EXAMPLES

```
(*min (mod!! (self-address!!) (!! 5)))  <=>  0
```

REFERENCES

See also the related global operators:

***and**	***integer–length**	***logand**	***logior**	
***logxor**	***max**	***or**	***sum**	***xor**

min!! [*Function*]

Determines in parallel the minimum numeric value of the supplied pvars.

SYNTAX

min!! *numeric–pvar* **&rest** *numeric–pvars*

ARGUMENTS

numeric–pvar, numeric–pvars
Non-complex numeric pvars. Pvars for which the minimum value is determined.

RETURNED VALUE

max–pvar Temporary numeric pvar. In each active processor, contains the minimum of the corresponding values of the supplied *numeric–pvar* arguments.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This returns a pvar that contains in each processor the minimum of the corresponding values of the supplied *numeric–pvars* in that processor.

EXAMPLES

```
(ppp (min!!   (mod!! (self-address!!) (!! 2))
              (mod!! (self-address!!) (!! 3))))  =>
0 1 0 0 0 1 0 1 0 0 0 1 0 1 . . .
```

---

minusp!! [Function]

Performs a parallel test for negative values on the supplied pvar.

SYNTAX

minusp!! *numeric-pvar*

ARGUMENTS

numeric-pvar Numeric pvar. Tested in parallel for negative values.

RETURNED VALUE

minusp-pvar Temporary boolean pvar. Contains the value **t** in each active processor where the corresponding value of *numeric-pvar* is negative. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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

NOTES

```
(minusp!! (!! -1))      <=>  t!!
(minusp!! (!! -0.0))    <=>    nil!!
```

mod!! *[Function]*

Performs a parallel modulo operation on the supplied pvars.

SYNTAX

mod!! *numeric–pvar divisor–pvar*

ARGUMENTS

numeric–pvar	Non-complex numeric pvar. Pvar for which modulo remainder is calculated.
divisor–pvar	Integer pvar. Pvar by which *numeric–pvar* is divided.

RETURNED VALUE

remainder–pvar	Temporary numeric pvar, of same type as *numeric–pvar*. In each active processor, contains the result of dividing the value of *numeric–pvar* modulo the value of *divisor–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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

EXAMPLES

```
(ppp (mod!! (self-address!!) (!! 5)))  =>
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 . . .
```

---

most–negative–float!!
most–positive–float!! [*Function*]

Return a pvar containing the floating-point value that is closest to negative or positive infinity in the format of the supplied floating-point pvar.

SYNTAX

most–negative–float!! *floating–point–pvar*
most–positive–float!! *floating–point–pvar*

ARGUMENTS

floating–point–pvar
Floating-point pvar. Determines format of returned pvar.

RETURNED VALUE

most–neg/pos–pvar Temporary floating-point pvar. In each active processor, contains the floating-point value closest to negative (or positive) infinity that is representable in the same format (single- or double-precision) as the corresponding value of *floating–point–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

These functions return a floating-point pvar with the same format (single- or double-precision) as the *floating–point–pvar* argument. In each processor, the returned value is the floating point number closest to negative (or positive) infinity.

EXAMPLES

The argument *floating-point-pvar* may be any floating point pvar of the required format. For example,

```
(most-negative-float!! (!! 0.0))  <=>  (!! -3.4028235e38)

(most-positive-float!! (!! 0.0))  <=>  (!! 3.4028235e38)
```

The same results would be obtained with any single-precision floating-point pvar argument.

REFERENCES

See also these related floating-point pvar limit functions:

float-epsilon!! **least-negative-float!!** **least-positive-float!!** **negative-float-epsilon!!**

negative–float–epsilon!! [Function]

Returns a pvar containing the smallest negative floating-point value representable in the format of the supplied floating-point pvar.

SYNTAX

negative–float–epsilon!! *floating–point–pvar*

ARGUMENTS

floating–point–pvar
Floating-point pvar. Determines format of returned pvar.

RETURNED VALUE

epsilon–pvar Temporary floating-point pvar. In each active processor, contains the smallest negative value representable in the same format as the corresponding value of *floating–point–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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.

REFERENCES

See also these related floating-point pvar limit functions:

float–epsilon!! **least–negative–float!!** **least–positive–float!!**
most–negative–float!! **most–positive–float!!**

*news [*Defun]

Performs grid (NEWS) communication, copying values from the source pvar to the destination pvar.

SYNTAX

***news** *source–pvar dest–pvar* **&rest** *relative–coordinate–integers*

ARGUMENTS

source–pvar	Pvar expression. Pvar from which values are copied.
dest–pvar	Pvar expression. Pvar in which values are stored.
relative–coordinate–integers	Series of integers. Specifies relative distance over which copy takes place along each dimension of the current VP set. The number of arguments must be equal to the rank of the current machine configuration.

RETURNED VALUE

nil	Executed for side effect only.

SIDE EFFECTS

Destructively alters *dest–pvar* to contain values from *source–pvar* transmitted across the grid.

DESCRIPTION

This function does near-neighbor store communication. Each active processor in the current VP set takes the value of *source–pvar* and stores it in the supplied *dest–pvar*, in the processor that is *relative–coordinate–integers* away across the *n*-dimensional grid of the current VP set.

The *source–pvar* argument is evaluated only by processors in the currently selected set, but the *dest–pvar* argument can be modified in any processor. In other words, even though only active processors transmit values from *source–pvar*, values can be received and stored in *dest–pvar* by any processor, active or not.

The *relative–coordinate–integer* arguments specify a single relative grid address used by all active processors in determining the address of the destination, i.e., if the *n*th *relative–coordinate–integer* argument is the value *j*, then each active processor will transmit a value to the processor *j* units away along dimension *n*.

The grid addresses calculated by a ***news** operation are toroidal, i.e., there are no upper or lower bounds on the values of the *relative–coordinate–integer* arguments. Where grid addresses are produced that specify processors off the edge of the current grid, those addresses wrap around to the opposite edge of the grid.

EXAMPLES

The ***news** macro can be used to perform global shifts of data across processor grids of any dimension. However, the macro is most commonly used on two-dimensional grids, where each processor has four neighbors, one each to the "left" and "right" along dimension 0, and one each "up" and "down" along dimension 1.

The following expressions define such a grid, along with two pvars that will be used in the following examples.

```
(*cold-boot :initial-dimensions '(32 16))
(*defvar source (random!! (!! 10)))
(*defvar dest)
```

A call to **ppp** displays the grid of values stored in the **source** pvar.

```
(ppp source :mode :grid :end '(4 4) :format "~2D ")

 7  9  8  6
 9  5  2  7
 6  2  4  2
 8  5  9  1
```

The following example of a call to ***news** shifts the entire grid over 1 to the right and down 1. Values are wrapped around from the right and lower edges to the left and upper edges.

```
(*news source dest 1 1)
(ppp dest :mode :grid :end '(4 4) :format "~2D ")

 8  5  8  1
 6  7  9  8
 8  9  5  2
 4  6  2  4
```

The next example shows that the value of the *dest–pvar* in unselected processors can be altered by a call to ***news**. The processors in the even columns, which are selected, send data to the processors in the odd columns, which are not selected. Even though the processors in the odd columns are deselected, they may still recieve and store values.

```
(*set dest (!! 0))
(*when (evenp!! (self-address-grid!! (!! 0)))
   (*news source dest 1 0))
(ppp dest :mode :grid :end '(4 4) :format "~2D ")

 0  7  0  8
 0  9  0  2
 0  6  0  4
 0  8  0  9
```

NOTES

Notice that ***news** is to **news!!** as ***pset** is to **pref!!**. Thus, while ***news** sends information to processors, **news!!** retrieves information from processors. Like ***news, news!!** assumes a toroidal arrangement of grid addresses, i.e., addresses wrap around the grid.

Performance Note:

Although seemingly symmetric, the CM-2 *Lisp implementation of **news!!** is faster than the CM-2 *Lisp implementation of ***news**.

Usage Note:

The grid address assigned to a processor by a one-dimensional VP set is not the same as the processor's send address. For example, given the one-dimensional grid defined by

```
(*cold-boot  :initial-dimensions
             (list *minimum-size-for-vp-set*))
```

the following expression displays in send address (**:mode :cube**) order the send addresses of a sample set of processors:

```
(ppp (self-address!!) :mode :cube :start 24 :end 40)

24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
```

and this expression displays the grid addresses of the same processors in send address order:

```
(ppp (self-address-grid!! (!! 0)) :mode :cube
                                  :start 24 :end 40)

24 25 26 27 28 29 30 31 48 49 50 51 52 53 54 55
```

Notice that the grid addresses of the last eight processors in this example are different from their send addresses. In general, there is no simple way to relate the grid address assigned to a processor by a VP to the send address of that processor except by the *Lisp address conversion functions **cube-from-grid-address**, **cube-from-vp-grid-address**, **grid-from-cube-address**, and **grid-from-vp-cube-address**. The assignment depends on such factors as the size and shape of the VP set, and on the number of physical processors attached.

Of course, if the grid addresses are displayed in grid address (**:mode :grid**) order, the addresses displayed will be sequential:

```
(ppp (self-address-grid!! (!! 0)) :mode :grid
                                  :start 24 :end 40)

24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
```

However, in this example, the processors for which the addresses are being displayed are not the same as in the previous two examples. Displaying processor grid addresses in grid address order by definition displays the addresses of those processors whose grid addresses are sequential.

The errors produced by neglecting this distinction are more pervasive than these examples demonstrate. For example, it is a common mistake to expect the expression

```
(ppp (news!! (self-address!!) 1)  :mode :cube
                                  :start 24 :end 40)
```

to display a series of sequential send addresses. In fact, it displays this:

```
24 25 26 27 28 29 30 31 48 33 34 35 36 37 38 39
```

The following expression produces the expected result:

```
(ppp (news!! (self-address-grid!! (!! 0)) 1)
      :mode :grid :start 24 :end 40)

24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
```

REFERENCES

See also these related NEWS communication operators:

news!!	**news–border!!**
***news–direction**	**news–direction!!**

See also these related off-grid processor address tests:

off–grid–border–p!!	**off–grid–border–relative–direction–p!!**
off–grid–border–relative–p!!	**off–vp–grid–border–p!!**

See also these related processor communication operators:

pref!!	***pset**

news!! [*Macro*]

Performs grid (NEWS) communication, returning a pvar containing values copied from the supplied pvar.

SYNTAX

news!! *source–pvar* **&rest** *relative–coordinate–integers*

ARGUMENTS

source–pvar Pvar expression. Pvar from which values are copied.

relative–coordinate–integers
Set of integers. Specifies relative distance over which copy takes place along each dimension of the current VP set. The number of arguments must be equal to the rank of the current machine configuration.

RETURNED VALUE

news–value–pvar Temporary pvar, of same type as *source–pvar*. In each active processor, contains a copy of the value of *source–pvar* from the processor specified by the set of *relative–coordinate–integers.*

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This macro does near-neighbor fetch communication. Each processor in the currently selected set retrieves the value of *source–pvar* from the processor that is *relative–coordinate–integers* away across the *n*-dimensional grid of the current VP set.

Even though only active processors retrieve values from *source–pvar*, values can be retrieved from any processor, not just those in the currently selected set. In other words, it is legal for the grid address specified by *relative–coordinate–integers* to cause values to be retrieved from processors that are not in the currently selected set.

The *relative–coordinate–integer* arguments specify a single relative grid address used by all active processors in determining the address of the destination, i.e., if the *n*th *relative–coordinate–integer* argument is the value *j*, then each active processor will retrieve a value from the processor *j* units away along dimension *n*.

The grid addresses calculated by a **news!!** operation are toroidal, i.e., there are no upper or lower bounds on the values of the *relative–coordinate–integer* arguments. Where grid addresses are produced that specify processors off the edge of the current grid, those addresses wrap around to the opposite edge of the grid.

EXAMPLES

The **news!!** macro can be used to perform global shifts of data across processor grids of any dimension. However, the macro is most commonly used on two-dimensional grids, where each processor has four neighbors, one each to the "left" and "right" along dimension 0, and one each "up" and "down" along dimension 1.

The following expressions define such a grid, along with two pvars that will be used in the following examples.

```
(*cold-boot :initial-dimensions '(32 16))
(*defvar source (random!! (!! 10)))
(*defvar dest (!! 0))
```

A call to **ppp** displays the grid of values stored in the **source** pvar.

```
(ppp source :mode :grid :end '(4 4) :format "~2D ")

 7  9  8  6
 9  5  2  7
 6  2  4  2
 8  5  9  1
```

The following example of a call to **news!!** shifts the entire grid over 1 to the left and up 1. Values are wrapped around from the left and upper edges to the right and bottom edges (not shown).

```
(ppp (news!! source 1 1) :mode :grid :end '(4 4)
                          :format "~2D ")

 5  2  7  4
 2  4  2  5
 5  9  1  3
 6  7  6  1
```

The next example shows that the value of the *source–pvar* in unselected processors can be retrieved by selected processsors during a call to **news!!**. The processors in the even columns, which are selected, retrieve data from the processors in the odd columns, which are not selected.

```
(*set dest (!! 0))

(*when (evenp!! (self-address-grid!! (!! 0)))
   (*set dest (news!! source 1 0)))

(ppp dest :mode :grid :end '(4 4) :format "~2D ")

9  0  6  0
5  0  7  0
2  0  2  0
5  0  1  0
```

The *source–pvar* argument to **news!!** is evaluated only in those processors from which data is being retrieved, not in the processors doing the retrieving. This means that operations signalling an error when the entire set of processors is selected may be perfectly legal when the currently selected set is restricted to a subset of processors. For example, consider the expression

```
(*when (evenp!! (self-address-grid!! (!! 0)))
  (*set dest
    (round!!
      (news!! (/!! (!! 24) (self-address-grid!! (!! 0)))
              1 0))))

(ppp dest :mode :grid :end '(4 4) :format "~2D ")

24  0  8  0
24  0  8  0
24  0  8  0
24  0  8  0
```

If the **/!!** operation in this example was performed with the entire set of processors selected, then a division by 0 would have occurred in the left-most column of processors because **(self–address–grid!! (!! 0))** returns 0 for each processor in that column. The division was actually performed only in the processors belonging to the odd columns, i.e., those processors having data retrieved from them, so no error was signalled.

NOTES

Notice that **news!!** is to ***news** as **pref!!** is to ***pset**. Thus, while **news!!** retrieves information from processors, ***news** sends information to processors. Like **news!!**, ***news** assumes a toroidal arrangement of grid addresses, i.e., addresses wrap around the grid.

Performance Notes:

Although seemingly symmetric, the CM-2 *Lisp implementation of **news!!** is faster than the CM-2 *Lisp implementation of ***news**.

Also, when **news!!** is invoked with relative coordinates that are powers of two, as in

```
(news!! pvar 8 16)
```

the CM-2 implementation of *Lisp uses special Paris instructions that are able to quickly retrieve the data. The above call to **news!!** is therefore signficantly faster than a call to **news!!** with non-power-of-two arguments, such as

```
(news!! pvar 7 15)
```

Usage Note:

The grid address assigned to a processor by a one-dimensional VP set is not the same as the processor's send address. For example, given the one-dimensional grid defined by

```
(*cold-boot   :initial-dimensions
              (list *minimum-size-for-vp-set*))
```

the following expression displays in send address (**:mode :cube**) order the send addresses of a sample set of processors

```
(ppp (self-address!!) :mode :cube :start 24 :end 40)

24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
```

and this expression displays the grid addresses of the same processors in send address order:

```
(ppp (self-address-grid!! (!! 0)) :mode :cube
                                  :start 24 :end 40)

24 25 26 27 28 29 30 31 48 49 50 51 52 53 54 55
```

Notice that the grid addresses of the last eight processors in this example are different from their send addresses. In general, there is no simple way to relate the grid address assigned to a processor by a VP to the send address of that processor except by the

*Lisp address conversion functions **cube-from-grid-address**, **cube-from-vp-grid-address**, **grid-from-cube-address**, and **grid-from-vp-cube-address**. The assignment depends on such factors as the size and shape of the VP set, and on the number of physical processors attached.

Of course, if the grid addresses are displayed in grid address (**:mode :grid**) order, the addresses displayed will be sequential:

```
(ppp (self-address-grid!! (!! 0)) :mode :grid
                                   :start 24 :end 40)

24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
```

However, in this example, the processors for which the addresses are being displayed are not the same as in the previous two examples. Displaying processor grid addresses in grid address order by definition displays the addresses of those processors whose grid addresses are sequential.

The errors produced by neglecting this distinction are more pervasive than these examples demonstrate. For example, it is a common mistake to expect the expression

```
(ppp (news!! (self-address!!) 1)  :mode :cube
                                   :start 24 :end 40)
```

to display a series of sequential send addresses. In fact, it displays this:

```
24 25 26 27 28 29 30 31 48 33 34 35 36 37 38 39
```

The following expression produces the expected result:

```
(ppp (news!! (self-address-grid!! (!! 0)) 1)
      :mode :grid :start 24 :end 40)

24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
```

REFERENCES

See also these related NEWS communication operators:

***news**	**news-border!!**
***news-direction**	**news-direction!!**

See also these related off-grid processor address tests:

off-grid-border-p!!	**off-grid-border-relative-direction-p!!**
off-grid-border-relative-p!!	**off-vp-grid-border-p!!**

See also these related processor communication operators:

pref!!	***pset**

news–border!! [*Macro*]

Performs grid (NEWS) communication, returning a pvar containing values copied from the supplied source pvar, with references off the grid satisfied by the supplied border pvar.

SYNTAX

news–border!! *source–pvar border–pvar* **&rest** *relative–coordinate–integers*

ARGUMENTS

source–pvar Pvar expression. Pvar from which values are copied.

border–pvar Pvar expression. Value returned for all references off the grid.

relative–coordinate–integers
Set of integers. Specifies relative distance over which copy takes place along each dimension of the current VP set. The number of arguments must be equal to the rank of the current machine configuration.

RETURNED VALUE

news–value–pvar Temporary pvar. In each active processor, contains a copy of the value of *source–pvar* in the processor specified by the set of *relative–coordinate–integers*, or the value of *border–pvar*, where the location specified is off the grid.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This macro performs the same operation as **news!!**, with the exception that, wherever a processor would be directed to retrieve a value from a location off the grid of the current VP set, the processor instead returns the value of the supplied *border–pvar*.

EXAMPLES

A sample call to **news–border!!** is

```
(news-border!! pvar border-pvar 1 1)
```

The **news–border!!** macro can be used to perform global shifts of data with a specific "boundary" value stored in all processors that attempt to read information from outside the boundaries of the grid. For example, given the two-dimensional grid configuration defined by

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

the expression

```
(ppp  (news-border!!
         (self-address-grid!! (!! 0)) (!! -1) -1 -1)
      :mode :grid
      :end '(4 4)
      :format "~2D ")
```

performs a diagonal shift of data "downwards" and "rightwards" across the grid, producing the following output:

```
-1 -1 -1 -1
-1  0  1  2
-1  0  1  2
-1  0  1  2
```

The value –1 is stored into processors along the "top" and "left" edges of the grid because these are the processors that attempt to read outside the grid in this operation.

REFERENCES

See also these related NEWS communication operators:

***news** **news!!**
***news–direction** **news–direction!!**

See also these related off-grid processor address tests:

off–grid–border–p!! **off–grid–border–relative–direction–p!!**
off–grid–border–relative–p!! **off–vp–grid–border–p!!**

See also these related processor communication operators:

pref!! ***pset**

*news–direction [*Defun]

Performs NEWS (grid) communication along a single dimension, copying values from the source pvar to the destination pvar.

SYNTAX

***news–direction** *source–pvar destination–pvar dimension–scalar distance–scalar*

ARGUMENTS

source–pvar	Pvar expression. Pvar from which values are copied.
destination–pvar	Pvar expression. Pvar into which values are stored.
dimension–scalar	Integer. Dimension along which to perform copy.
distance–scalar	Integer. Distance over which values are copied.

RETURNED VALUE

nil	Executed for side effect only.

SIDE EFFECTS

Destructively alters *destination–pvar* to contain values from *source–pvar* transmitted across the NEWS grid.

DESCRIPTION

Performs a ***news** operation on the source pvar, along the specified dimension and at the specified distance. Each active processor in the current VP set sends *source–pvar* data to the processor that is *distance–scalar* processors away along the *dimension–scalar* axis, and stores it in *destination–pvar.*

The *source–pvar* and *destination–pvar* parameters must both be in the current VP set.

The *source–pvar* argument is evaluated only by processors in the currently selected set, but the *destination–pvar* argument can be modified in any processor. In other words, even though only active processors transmit values from *source–pvar*, values can be received and stored in *destination–pvar* by any processor, not just those in the currently selected set.

The *dimension–scalar* parameter must be an integer in the range [0..($N-1$)], where N is the number of dimensions defined for the current VP set.

The *distance–scalar* parameter must be an integer. The sign of this value determines in which direction along the specified dimension data is sent. Grid addresses wrap around where necessary.

This function permits ***news** operations along a given dimension without requiring specification of the total number of dimensions in the current VP set. Thus, assuming a three-dimensional machine configuration,

```
(*news-direction my-pvar my-result 2 3)
<=>
(*news my-pvar my-result 0 0 3)
```

EXAMPLES

This function is particularly useful when writing subroutines that must do NEWS operations along a particular dimension of the currently defined grid but may be called with VP sets of differing ranks active.

```
(defun shift-upward-along-y-axis  (dest-pvar
                                   source-pvar
                                   distance)
  (*news-direction source-pvar dest-pvar 1 (- distance)))
```

REFERENCES

See also these related NEWS communication operators:

***news** **news!!** **news–border!!**
news–direction!!

See also these related off-grid processor address tests:

off–grid–border–p!! **off–grid–border–relative–direction–p!!**
off–grid–border–relative–p!! **off–vp–grid–border–p!!**

See also these related processor communication operators:

pref!! ***pset**

news–direction!! [*Macro*]

Performs NEWS (grid) communication along a specified dimension, returning a pvar containing values copied from the supplied pvar.

SYNTAX

news–direction!! *source–pvar dimension–scalar distance–scalar*

ARGUMENTS

source–pvar	Pvar expression. Pvar from which values are copied.
dimension–scalar	Integer. Dimension along which to perform copy.
distance–scalar	Integer. Distance over which values are copied.

RETURNED VALUE

news–value–pvar	Temporary pvar, of same type as *source–pvar*. In each active processor, contains a copy of the value of *source–pvar* in the processor *distance–scalar* away along the dimension specified by *dimension–scalar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

Performs a **news!!** operation on the specified pvar, along the specified dimension and at the specified distance. Each active processor in the current VP set retrieves *source–pvar* data from the processor that is *distance–scalar* processors away along the *dimension–scalar* axis.

The *source–pvar* parameter must be in the current VP set.

Even though only active processors retrieve values from *source–pvar*, values can be retrieved from any processor, not just those in the currently selected set. In other words, it is legal for the grid address specified by *dimension–scalar* and *distance–scalar* to cause values to be retrieved from processors that are not in the currently selected set.

The *dimension–scalar* parameter must be an integer in the range $[0..(N-1)]$, where N is the number of dimensions defined for the current VP set.

The *distance–scalar* parameter must be an integer. The sign of this value determines from which direction along the specified dimension data is retrieved. Grid addresses wrap around where necessary.

This function permits **news!!** operations along a given dimension without requiring specification of the total number of dimensions in the current VP set. Thus, assuming a three-dimensional machine configuration has been defined, the following equivalence holds:

```
(news-direction!! my-pvar 1 2)
<=>
(news!! my-pvar 0 2 0)
```

EXAMPLES

This function is particularly useful when writing subroutines that must do NEWS operations along a particular dimension of the currently defined grid but may be called with VP sets of differing ranks active.

```
(defun shift-upward-along-y-axis (pvar distance)
  (news-direction!! pvar 1 distance)))
```

REFERENCES

See also these related NEWS communication operators:

***news** **news!!** **news–border!!**
***news–direction**

See also these related off-grid processor address tests:

off–grid–border–p!! **off–grid–border–relative–direction–p!!**
off–grid–border–relative–p!! **off–vp–grid–border–p!!**

See also these related processor communication operators:

pref!! ***pset**

next–power–of–two–>= [*Function*]

Returns the next power of two greater than or equal to the supplied integer.

SYNTAX

next–power–of–two–>= *positive–integer*

ARGUMENTS

positive–integer	Value for which the next higher power of two is determined.

RETURNED VALUE

power–of–two	Integer. Next power of two greater than or equal to *positive–integer.*

SIDE EFFECTS

None.

DESCRIPTION

This function returns the first consecutive integer satisfying **power–of–two–p** that is greater than or equal to *positive–integer.*

EXAMPLES

```
(next-power-of-two->= 356)  =>  512
```

NOTES

Usage Note:

This function is useful in computing the dimensions of VP sets, because each dimension of a VP set must be an integral power of two in size, and the total number of processors in a VP set must be a power of two multiple of the number of physical processors available.

For instance, if a data file has 23,432 items, a call to **next–power–of–two–>=**, specifically

```
(next-power-of-two->= 23432)   =>   32768
```

can be used to determine that a VP set of size 32768 is required to process the data.

REFERENCES

See also the related predicate **power–of–two–p**.

The **next–power–of–two–>=** function is most useful in combination with the following VP set definition operators:

def–vp–set **create–vp–set** **let–vp–set**

not!! [Function]

Performs a parallel logical negation on the supplied pvar.

SYNTAX

not!! *pvar*

ARGUMENTS

pvar	Pvar expression. Pvar for which the logical negation is determined.

RETURNED VALUE

not–pvar	Temporary boolean pvar. Contains **t** in those active processors where *pvar* contains the value **nil**. Contains **nil** in all other processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This returns **t** for all processors in which *pvar* is **nil**, and **nil** otherwise.

REFERENCES

See also the related global operators:

***and**	***integer–length**	***logand**
***logior**	***logxor**	***max**
***min**	***or**	***sum**
***xor**		

See also the related logical operators:

and!!	**or!!**	**xor!!**

---

notany!! *[Function]*

Tests in parallel whether the supplied pvar predicate is false for every set of elements having the same indices in the supplied sequence pvars.

SYNTAX

notany!! *predicate sequence–pvar* **&rest** *sequence–pvars*

ARGUMENTS

predicate — Boolean pvar predicate. Used to test elements of sequences in the *sequence–pvar* arguments. Must take as many arguments as the number of *sequence–pvar* arguments supplied.

sequence–pvar, sequence–pvars — Sequence pvars. Pvars containing, in each processor, sequences to be tested by *predicate.*

RETURNED VALUE

notany–pvar — Temporary boolean pvar. Contains the value **t** in each active processor in which every set of elements taken from the sequences of the *sequence–pvars* fails the *predicate.* Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The **notany!!** function returns a boolean pvar indicating in each processor whether the supplied *predicate* is false for every set of elements with the same indices in the sequences of the supplied *sequence–pvars.*

In each processor, the *predicate* is first applied to the index 0 elements of the sequences in the *sequence–pvars*, then to the index 1 elements, and so on. The *n*th time *predicate*

is called, it is applied to the *nth* element of each of the sequences. If *predicate* returns **t** in any processor, that processor is temporarily removed from the currently selected set for the remainder of the operation. The operation continues until the shortest of the *sequence–pvars* is exhausted, or until no processors remain selected.

The pvar returned by **notany!!** contains **t** in each processor where *predicate* returns the value **nil** for every set of sequence elements. If *predicate* returns **t** for any set of sequence elements in a given processor, **notany!!** returns **nil** in that processor.

EXAMPLES

```
(notany!! 'equalp!! (!! #(1 2 3)) (!! #(9 4 1)))  <=>  t!!
```

NOTES

Compiler Note:

The *Lisp compiler does not compile this operation.

REFERENCES

See the related functions **every!!**, **notevery!!**, and **some!!**.

See also the general mapping function **amap!!**.

notevery!! [*Function*]

Tests in parallel whether the supplied pvar predicate is false for at least one set of elements having the same indices in the supplied sequence pvars.

SYNTAX

notevery!! *predicate sequence–pvar* **&rest** *sequence–pvars*

ARGUMENTS

predicate — Boolean pvar predicate. Used to test elements of sequences in the *sequence–pvar* arguments. Must take as many arguments as the number of *sequence–pvar* arguments supplied.

sequence–pvar, sequence–pvars — Sequence pvars. Pvars containing, in each processor, sequences to be tested by *predicate.*

RETURNED VALUE

notevery–pvar — Temporary boolean pvar. Contains the value **t** in each active processor in which at least one set of elements having the same indices in the sequences of the *sequence–pvars* fails the *predicate*. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The **notevery!!** function returns a boolean pvar indicating in each processor whether the supplied *predicate* is false for at least one set of elements with the same indices in the sequences of the supplied *sequence–pvars.*

In each processor, the *predicate* is first applied to the index 0 elements of the sequences in the *sequence–pvars*, then to the index 1 elements, and so on. The *n*th time *predicate*

is called, it is applied to the *nth* element of each of the sequences. If *predicate* returns **nil** in any processor, that processor is temporarily removed from the currently selected set for the remainder of the operation. The operation continues until the shortest of the *sequence–pvars* is exhausted, or until no processors remain selected.

The pvar returned by **notevery!!** contains **t** in each processor where *predicate* returns the value **nil** for at least one set of sequence elements. If *predicate* returns **t** for every set of sequence elements in a given processor, **notevery!!** returns **nil** in that processor.

EXAMPLES

```
(notevery!! 'equalp!! (!! #(1 2 3)) (!! #(1 2 4)))  <=>  t!!
```

NOTES

Compiler Note:

The *Lisp compiler does not compile this operation.

REFERENCES

See the related functions **every!!**, **notany!!**, and **some!!**.

See also the general mapping function **amap!!**.

*nreverse [*Defun]

Destructively reverses each sequence stored in the supplied sequence pvar.

SYNTAX

***nreverse** *sequence–pvar*

ARGUMENTS

sequence–pvar Sequence pvar. Pvar containing sequences to be reversed.

RETURNED VALUE

sequence–pvar Sequence pvar. The supplied *sequence–pvar* with each of its sequences destructively reversed.

SIDE EFFECTS

None.

DESCRIPTION

The function ***nreverse** destructively modifies *sequence–pvar* to contain its elements in reverse order. The argument *sequence–pvar* must be a vector pvar.

EXAMPLES

```
(*nreverse (!! #(1 2 3 4))) <=> (!! #(4 3 2 1))
```

NOTES

Compiler Note:

The *Lisp compiler does not compile this operation.

REFERENCES

See also these related *Lisp sequence operators:

copy-seq!!	***fill**	**length!!**
reduce!!	**reverse!!**	**subseq!!**

See also the generalized array mapping functions **amap!!** and ***map**.

nsubstitute!!, nsubstitute–if!!, nsubstitute–if–not!! *[Function]*

Performs a destructive parallel substitution operation on a sequence pvar, replacing specified old items with new items.

SYNTAX

nsubstitute!! *new–item old–item sequence–pvar*
&key **:test :test–not**
:start :end :key :from–end :count

nsubstitute–if!! *new–item test sequence–pvar*
&key **:start :end :key :from–end :count**

nsubstitute–if–not!! *new–item test sequence–pvar*
&key **:start :end :key :from–end :count**

ARGUMENTS

new–item — Pvar expression, of same data type as *sequence–pvar*. Item to substitute for *old–item* in each processor.

old–item — Pvar expression, of same data type as *sequence–pvar*. Item to be replaced in each processor.

test — One-argument pvar predicate. Test used in comparisons. Indicates a match by returning a non-**nil** result. Defaults to **eql!!**.

sequence–pvar — Sequence pvar. Pvar containing sequences to be modified.

:test — Two-argument pvar predicate. Test used in comparisons. Indicates a match by returning a non-**nil** result. Defaults to **eql!!**.

:test–not — Two-argument pvar predicate. Test used in comparisons. Indicates a match by returning a **nil** result.

:start — Integer pvar. Index of sequence element at which substitution starts in each processor. If not specified, search begins with first element. Zero-based.

:end — Integer pvar. Index of sequence element at which substitution ends in each processor. If not specified, search continues to end of sequence. Zero-based.

:key	One-argument pvar accessor function. Applied to *sequence–pvar* before search is performed.
:from–end	Boolean. Whether to begin substitution from end of sequence in each processor. Defaults to **nil**.
:count	Integer pvar. Maximum number of replacements to perform in each processor. Defaults to **(length!!** *sequence–pvar***)**

RETURNED VALUE

sequence–pvar	Sequence pvar. The supplied *sequence–pvar* with each of its sequences destructively modified.

SIDE EFFECTS

Destructively modifies *sequence–pvar*, replacing elements matching *old–item* with copies of *new–item*.

DESCRIPTION

These functions are the parallel equivalent of the Common Lisp **nsubstitute** functions, and are the destructive counterparts of the non-destructive **substitute!!** functions.

In each processor, the function **nsubstitute!!** searches *sequence–pvar* for elements that match *old–item*. Each such element is destructively modified to contain the value specified by *new–item*. Elements of *sequence–pvar* are tested against *old–item* with the **eql!!** operator unless another comparison operator is supplied as either of the **:test** or **:test–not** arguments. The keywords **:test** and **:test–not** may not be used together. A lambda form that takes two pvar arguments and returns a boolean pvar result may be supplied as either the **:test** and **:test–not** argument.

The function **nsubstitute–if!!** searches *sequence–pvar* for elements satisfying *test*. Each such element is destructively modified to contain the value specified by *new–item*. A lambda form that takes a single pvar argument and returns a boolean pvar result may be supplied as the *test* argument.

The function **nsubstitute–if–not!!** similarly searches *sequence–pvar* for elements failing *test*.

The keyword **:from–end** takes a boolean pvar that specifies from which end of *sequence–pvar* in each processor the operation will take place.

Arguments to the keywords **:start** and **:end** define a subsequence to be operated on in each processor.

The **:key** keyword accepts a user-defined function used to extract a search key from *sequence–pvar*. This key function must take one argument: an element of *sequence–pvar*.

The **:count** keyword argument must be a positive integer pvar with values less than or equal to **(length!!** *sequence–pvar***)**. In each processor at most *count* elements are substituted.

NOTES

Compiler Note:

The *Lisp compiler does not compile this operation.

REFERENCES

This function is one of a group of similar sequence operators, listed below:

count!!	**count–if!!**	**count–if–not!!**
find!!	**find–if!!**	**find–if–not!!**
nsubstitute!!	**nsubstitute–if!!**	**nsubstitute–if–not!!**
position!!	**position–if!!**	**position–if–not!!**
substitute!!	**substitute–if!!**	**substitute–if–not!!**

See also the generalized array mapping functions **amap!!** and ***map**.

null!! *[Function]*

Performs a parallel test for **nil** values on the supplied pvar.

SYNTAX

null!! *pvar*

ARGUMENTS

pvar	Pvar expression. Pvar to be tested for **nil** values.

RETURNED VALUE

null–pvar	Temporary boolean pvar. Contains **t** in those active processors where *pvar* contains the value **nil**. Contains **nil** in all other processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function is functionally equivalent to **not!!**.

---

numberp!! [*Function*]

Performs a parallel test for numeric values on the supplied pvar.

SYNTAX

numberp!! *pvar*

ARGUMENTS

pvar	Pvar expression. Pvar to be tested for numeric values.

RETURNED VALUE

numberp–pvar	Temporary boolean pvar. Contains **t** in those active processors where *pvar* contains a numeric value. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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

REFERENCES

See also these related pvar data type predicates:

booleanp!! **characterp!!** **complexp!!**
floatp!! **front–end–p!!** **integerp!!**
string–char–p!! **structurep!!**
typep!!

oddp!! *[Function]*

Performs a parallel test for odd values on the supplied integer pvar.

SYNTAX

oddp!! *integer–pvar*

ARGUMENTS

integer–pvar Integer pvar. Pvar to be tested for odd values.

RETURNED VALUE

oddp–pvar Temporary boolean pvar. Contains **t** in each active processor where *integer–pvar* contains an odd value. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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

off–grid–border–p!! [*Function*]

Performs a parallel test on the supplied pvar(s) for grid (NEWS) addresses that are outside the currently specified grid dimensions.

SYNTAX

off–grid–border–p!! *coordinate–pvar* **&rest** *coordinate–pvars*

ARGUMENTS

coordinate–pvar, coordinate–pvars

Integer pvars. Pvars specifying a grid (NEWS) address in each processor. The number of arguments must be equal to the rank of the current machine configuration.

RETURNED VALUE

off–gridp–pvar Temporary boolean pvar. Contains the value **t** in each active processor where the corresponding values of the *coordinate–pvars* specify a location outside the currently specified grid dimensions. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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 *coordinate–pvar* arguments. This function determines whether or not these grid addresses are within the bounds defined by the current VP set.

This function returns a boolean pvar that has the value **t** in each processor where the supplied *coordinate–pvars* specify a grid address that is invalid given the current grid dimensions, and **nil** otherwise.

EXAMPLES

This example defines a two-dimensional grid configuration, and generates a pair of pvars that contain random grid addresses.

```
(*cold-boot :initial-dimensions '(4 4))
(*defvar x-coordinate (random!! (!! 6)))
(*defvar y-coordinate (random!! (!! 6)))

(ppp x-coordinate :mode :grid)

4 5 5 5
4 2 2 2
2 1 5 3
5 1 2 3

(ppp y-coordinate :mode :grid)

0 1 0 5
0 0 2 4
1 1 4 4
5 3 1 1
```

Some of the grid addresses specified by the pvars will lie outside the grid of the VP set. A call to **off-grid-border-p!!** determines which grid addresses actually do lie outside the grid.

```
(ppp  (off-grid-border-p!! x-coordinate y-coordinate)
      :mode :grid :format "~3S ")
T   T   T   T
T   NIL NIL T
NIL NIL T   T
T   NIL NIL NIL
```

REFERENCES

This function tests whether the supplied grid addresses are within the grid dimensions of the current VP set. See the related function **off-vp-grid-border-p!!** for a way to test grid addresses in VP sets other than the current one.

See also these related NEWS communication operators:

***news**	**news!!**	**news-border!!**
***news-direction**	**news-direction!!**	

See also these related off-grid processor address tests:

off–grid–border–relative–direction–p!!
off–grid–border–relative–p!!

See also these related processor communication operators:

pref!! ***pset**

off-grid-border-relative-direction-p!! [*Function*]

Performs a parallel test for processors that access a location beyond the boundaries of the currently specified grid along the specified dimension.

SYNTAX

off-grid-border-relative-direction-p!! *dimension-scalar distance-scalar*

ARGUMENTS

dimension-scalar	Integer. Dimension along which to test references.
distance-scalar	Integer. Distance along dimension to test.

RETURNED VALUE

off-gridp-pvar	Temporary boolean pvar. Contains the value **t** in each active processor for which *distance-scalar* represents an access along the dimension specified by *dimension-scalar* that is beyond the boundary of the currently specified grid. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

Tests the relative grid addresses indicated by the specified *dimension-scalar* and *distance-scalar* for validity. A boolean pvar is returned.

The *dimension-scalar* argument must be an integer that is in the range $[0..(N-1)]$, where N is the number of dimensions defined for the current VP set.

The *distance-scalar* argument must be an integer and may be negative. The sign of this value determines in which direction along the specified dimension relative addresses are calculated.

The return value of this function is a boolean pvar that contains **t** in each processor for which an invalid relative address is specified and **nil** elsewhere.

If, for an active processor P in the current VP set, there exists another processor that is *distance–scalar* processors away along the *dimension–scalar* axis, then the result returned in processor P is **nil**.

EXAMPLES

This function is similar to **off–grid–border–p!!** and **off–grid–border–relative–p!!**. However, it permits relative address verification along a single dimension without requiring specification of the total number of dimensions in the current VP set. Thus, the following forms are equivalent,

```
(off-grid-border-relative-direction-p!! 1 5)
<=>
(off-grid-border-relative-p!! 0 5 0)
```

assuming a three-dimensional machine configuration.

REFERENCES

See also these related NEWS communication operators:

***news**	**news!!**	**news–border!!**
***news–direction**	**news–direction!!**	

See also these related off-grid processor address tests:

off–grid–border–p!!	
off–grid–border–relative–p!!	**off–vp–grid–border–p!!**

See also these related processor communication operators:

pref!!	***pset**

off-grid-border-relative-p!! [*Function*]

Performs a parallel test on the supplied pvar(s) for relative grid (NEWS) addresses that are outside the currently specified grid dimensions.

SYNTAX

off-grid-border-relative-p!! *relative-coord-pvar* **&rest** *relative-coord-pvars*

ARGUMENTS

relative-coord-pvar, relative-coord-pvars
Integer pvars. Pvars specifying a relative grid (NEWS) address in each processor. The number of arguments must be equal to the rank of the current machine configuration.

RETURNED VALUE

off-gridp-pvar Temporary boolean pvar. Contains the value **t** in each active processor where the corresponding values of the *relative-coord-pvars* specify a location outside the currently specified grid dimensions. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function tests relative grid addresses for validity. In each processor, the *relative-coord-pvar* arguments specify a relative grid address. Specifically, the jth *relative-coord-pvar* argument specifies for each processor the distance between that processor and the processor to be referenced, along the jth dimension. The **off-grid-border-relative-p!!** function determines whether or not the relative grid address in each processor is within the bounds of the current grid configuration.

EXAMPLES

This example defines a two-dimensional grid configuration, and then makes a call to **off–grid–border–relative–p!!** that tests the same relative grid address, (–1,–1), in each processor. As the result of this operation shows, the only processors for which this relative grid address is off the edge of the grid are those processors on the "top" and "left" edges of the grid.

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

  (ppp (off-grid-border-relative-p!! (!! -1) (!! -1))
        :mode :grid :end '(4 4) :format "~3S ")

T   T   T   T
T   NIL NIL NIL
T   NIL NIL NIL
T   NIL NIL NIL
```

The **off–grid–border–relative–p!!** function can also be used to easily select all processors within two processors of the border.

```
(*when (or!!  (off-grid-border-relative-p!! (!! -2) (!! -2))
              (off-grid-border-relative-p!! (!!  2) (!!  2)))
  (check-border-condition))
```

REFERENCES

The **off–grid–border–relative–p!!** function is similar to **off–grid–border–p!!** except that the *relative–coord–pvars* specify relative grid addresses rather than absolute addresses.

See also these related NEWS communication operators:

***news** **news!!** **news–border!!**
***news–direction** **news–direction!!**

See also these related off-grid processor address tests:

off–grid–border–p!! **off–grid–border–relative–direction–p!!**
off–vp–grid–border–p!!

See also these related processor communication operators:

pref!! ***pset**

off–vp–grid–border–p!! [Function]

Performs a parallel test on the supplied pvar(s) for grid (NEWS) addresses that are outside the grid dimensions of the supplied VP set.

SYNTAX

off–vp–grid–border–p!! *vp–set coordinate–pvar* **&rest** *coordinate–pvars*

ARGUMENTS

vp–set VP set object. The grid addresses specified by the supplied *coordinate–pvars* are tested to determine whether they are within the grid boundaries of *vp–set*.

coordinate–pvar, coordinate–pvars

Integer pvars. Pvars that specify a grid (NEWS) address in each processor. The number of *coordinate–pvar* arguments must be equal to the number of dimensions in *vp–set*.

RETURNED VALUE

off–gridp–pvar Temporary boolean pvar. Contains the value **t** in each active processor where the corresponding values of the *coordinate–pvars* specify a location outside the grid dimensions of *vp–set*. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function tests grid addresses for validity relative to a specified VP set.

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 *coordinate–pvars* specify an invalid grid address. In all other processors, **nil** is returned.

EXAMPLES

This example creates a two-dimensional VP set, **two–d–vp–set**, a one-dimensional VP set, **my–vp–set**, and a pair of pvars belonging to **my–vp–set** that contain random grid addresses within **two–d–vp–set**.

```
(def-vp-set two-d-vp-set '(4 4))
(def-vp-set my-vp-set '(8))

(*defvar y-coordinate (random!! (!! 5)) nil my-vp-set))
(*defvar x-coordinate (random!! (!! 5)) nil my-vp-set))

(ppp x-coordinate)
1 4 1 3 0 0 3 1

(ppp y-coordinate)
4 0 2 2 3 1 1 4
```

A call to **off–grid–border–p!!**, specifically

```
(*with-vp-set my-vp-set
   (ppp (off-vp-grid-border-p!! two-d-vp-set
            x-coordinate y-coordinate)))
T T NIL NIL NIL NIL NIL T
```

demonstrates that the coordinate pairs contained in processors 0, 1, and 7 of the two coordinate pvars are invalid for **two–d–vp–set.**

As this example shows, it is not necessary for the *coordinate–pvar* arguments to belong to the specified *vp–set*, or to even have the same size (number of elements).

REFERENCES

This function is similar to **off–grid–border–p!!** except that it permits testing of grid addresses within a specific VP set other than the current one.

See also these related NEWS communication operators:

***news** **news!!** **news–border!!**
***news–direction** **news–direction!!**

See also these related off-grid processor address tests:

off–grid–border–p!! **off–grid–border–relative–direction–p!!**
off–grid–border–relative–p!!

See also these related processor communication operators:

pref!! ***pset**

*or [*Defun]

Takes the logical inclusive OR of all values in a pvar, returning a scalar value.

SYNTAX

***or** *pvar–expression*

ARGUMENTS

pvar–expression	Pvar expression. Pvar to which global inclusive OR is applied.

RETURNED VALUE

or–scalar	Scalar boolean value. The logical inclusive OR of the values in *pvar*.

SIDE EFFECTS

None.

DESCRIPTION

The ***or** function is a global operator. It returns a scalar value of **t** if the value of *pvar–expression* in any active processor is non-**nil**, and returns **nil** otherwise.

If there are no active processors, this function returns **nil**.

EXAMPLES

Two examples of the use of global operators such as ***or** are

```
(*defun =t!! (pvar) (not (*or (not!! pvar))))
(*defun =nil!! (pvar) (not (*or pvar)))
```

NOTES

To determine whether there are any processors currently selected, a handy idiom is

```
(*or t!!)
```

which returns **t** only if there are selected processors.

REFERENCES

See also the related global operators:

***and**	***integer–length**	***logand**
***logior**	***logxor**	***max**
***min**	***sum**	***xor**

See also the related logical operators:

and!!	**not!!**	**or!!**	**xor!!**

or!! [*Macro*]

Performs a parallel logical inclusive OR operation in all active processors.

SYNTAX

or!! &rest *pvar–exprs*

ARGUMENTS

pvar–exprs	Pvar expressions. Pvars to which parallel inclusive OR is applied.

RETURNED VALUE

or–pvar	Temporary boolean pvar. Contains in each active processor the logical inclusive OR of the corresponding values of the *pvar–exprs*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The **or!!** function performs a parallel logical inclusive OR operation. It evaluates each of the supplied *pvar–exprs* in order, from left to right, in all active processors. As soon as one of the *pvar–exprs* evaluates to a non-**nil** value in a processor, that processor is removed from the currently selected set for the remainder of the **or!!**.

The temporary pvar returned by **or!!** contains the value of the last of the *pvar–exprs* evaluated in each processor. If no *pvar–exprs* are supplied, the pvar **nil!!** is returned.

The function **or!!** provides a functionality for boolean pvars similar to that provided by the Common Lisp function **or** for boolean values.

EXAMPLES

A simple example of the use of the **or!!** macro is

```
(ppp     (or!!  (evenp!! (self-address!!))
                (<!! (self-address!! (!! 3))))
         :end 10)

T T T NIL T NIL T NIL T NIL
```

NOTES

Language Note:

Remember that **or!!** changes the currently selected set as it evaluates its arguments. This can have unwanted side effects in code that depends on unchanging selected sets, particularly code involving communication operators, such as **scan!!**.

For example, the expressions

```
(ppp (or!! (evenp!! (self-address!!))
           (<!! (scan!! (self-address!!) '+!!) (!! 5)))
     :end 8)

T T T T T NIL T NIL

(ppp (or!! (<!! (scan!! (self-address!!) '+!!) (!! 5))
            (evenp!! (self-address!!)))
     :end 8)

T T T NIL T NIL T NIL
```

exemplify a case in which using **or!!** may cause a non-intuitive result because of its deselection properties. In the first expression, the **scan!!** operation is performed only in the odd processors. In the second expression, the **scan!!** operation is performed in all processors, resulting in different set of displayed values.

This is the result of **or!!** deselecting those processors that satisfy any clause, before executing the next clause. One can avoid this in the following manner:

```
(*let ((b1 (evenp!! (self-address!!)))
       (b2 (<!! (scan!! (self-address!!) '+!!) (!! 3))))
  (declare (type boolean-pvar b1 b2))
  (or!! b1 b2))
```

REFERENCES

See also the related global operators:

***and**	***integer–length**	***logand**
***logior**	***logxor**	***max**
***min**	***or**	***sum**
***xor**		

See also the related logical operators:

and!!	**not!!**	**xor!!**

phase!! *[Function]*

Returns a pvar containing the phase angle of the supplied complex pvar.

SYNTAX

phase!! *numeric–pvar*

ARGUMENTS

numeric–pvar	Numeric pvar. Pvar for which the phase angle is calculated.

RETURNED VALUE

phase–ang–pvar	Numeric pvar. In each active processor, contains the phase angle of the corresponding complex value in *numeric–pvar.*

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function returns a temporary pvar containing in each processor the phase angle, in radians, of the complex value in *numeric–pvar.* Note: *numeric–pvar* need not explicitly contain complex values. Non-complex values are coerced to complex values with a zero imaginary component.

REFERENCES

See also these related complex pvar operators:

abs!!	**cis!!**	**complex!!**
conjugate!!	**imagpart!!**	**realpart!!**

---

plusp!! *[Function]*

Performs a parallel test for positive values on the supplied pvar.

SYNTAX

plusp!! *numeric–pvar*

ARGUMENTS

numeric–pvar	Numeric pvar. Tested in parallel for positive values.

RETURNED VALUE

plusp–pvar	Temporary boolean pvar. Contains the value **t** in each active processor where the corresponding value of *numeric–pvar* is positive. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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

---

position!!, position–if!!, position–if–not!! [*Function*]

Performs a parallel search on a sequence pvar, returning in each processor the positional index of the first sequence element matching the supplied item or passing/failing a test.

SYNTAX

position!!	*item–pvar sequence–pvar*	**&key :test :test–not :start :end :key :from–end**
position–if!!	*test sequence–pvar* **&key**	**:start :end :key :from–end**
position–if–not!!	*test sequence–pvar* **&key**	**:start :end :key :from–end**

ARGUMENTS

item–pvar	Pvar expression. Item to match in the corresponding value of *sequence–pvar*. Must be of the same data type as the elements of *sequence–pvar*.
test	One-argument pvar predicate. Used to test elements of *sequence–pvar*.
sequence–pvar	Sequence pvar. Contains sequences to be searched.
:test	Two-argument pvar predicate. Test used in comparisons. Indicates a match by returning a non-**nil** result. Defaults to **eql!!**.
:test–not	Two-argument pvar predicate. Test used in comparisons. Indicates a match by returning a **nil** result.
:start	Integer pvar. Index, zero-based, of sequence element at which search starts in each processor. If not specified, search begins with first element.
:end	Integer pvar. Index, zero-based, of sequence element at which search ends in each processor. If not specified, search continues to end of sequence.
:key	Oon-argument pvar accessor function. Applied to each element in *sequence–pvar* before test is performed.

:from-end	Boolean. Whether to begin search from end of sequence in all processors. Defaults to **nil**.

RETURNED VALUE

position-pvar	Temporary pvar, of same data type as elements of *sequence-pvar*. In each active processor, contains the numeric index of the first matching element of *sequence-pvar*. Returns the value –1 in processors where no match was found.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

These functions are the parallel equivalents of the Common Lisp **position** functions.

In each processor, the function **position!!** searches *sequence-pvar* for elements that match *item-pvar*. It returns a pvar containing the index of the first match found in each processor. In any processor failing the search, the returned pvar contains –1. Elements of *sequence-pvar* are tested against *item-pvar* with the **eql!!** operator unless another comparison operator is supplied as either of the **:test** or **:test-not** arguments. The keywords **:test** and **:test-not** may not be used together. A lambda form that takes two pvar arguments and returns a boolean pvar result may be supplied as either the **:test** and **:test-not** argument.

The function **position-if!!** searches *sequence-pvar* for elements that satisfy the supplied *test*. It returns a pvar containing the index of the first such element found in each processor. In any processor failing the search, the returned pvar contains –1. A lambda form that takes a single pvar argument and returns a boolean pvar result may be supplied as the *test* argument. Similarly, the function **position-if-not!!** searches *sequence-pvar* for elements that fail the supplied *test*.

Arguments to the keywords **:start** and **:end** define a subsequence to be operated on in each processor.

The **:key** argument specifies a one-argument pvar function that is applied in parallel to each element of *sequence-pvar* before the comparison with *item-pvar* is performed. This argument can be used to select a key value from a structure, or to manipulate the values being compared.

The keyword **:from–end** takes a boolean pvar that specifies from which end of *sequence–pvar* in each processor the operation will take place.

EXAMPLES

```
(*defvar vector-pvar (!! #(1 2 3 4 5 6 7)))

(position!! (!! 4) vector-pvar)    <=> (!! 3)
(position!! (!! 4) vector-pvar
          :test '=!! :key '1-!!) <=> (!! 4)
```

NOTES

Compiler Note:

The *Lisp compiler does not compile this operation.

REFERENCES

The functions **position!!**, **position–if!!**, and **position–if–not!!** are similar to the **find!!** functions. Here, however, it is the indices of the matching elements, rather than the elements themselves, that are returned.

These functions are members of a group of similar sequence operators, listed below:

count!!	**count–if!!**	**count–if–not!!**
find!!	**find–if!!**	**find–if–not!!**
nsubstitute!!	**nsubstitute–if!!**	**nsubstitute–if–not!!**
position!!	**position–if!!**	**position–if–not!!**
substitute!!	**substitute–if!!**	**substitute–if–not!!**

See also the generalized array mapping functions **amap!!** and ***map**.

power–of–two–p [*Function*]

Tests whether the supplied integer is an integral power of two.

SYNTAX

power–of–two–p *positive–integer*

ARGUMENTS

positive–integer Integer. Positive integer to be tested.

RETURNED VALUE

power–of–two–p Scalar boolean value. The value **t** if *positive–integer* is an integral power of two, and **nil** otherwise.

SIDE EFFECTS

None.

DESCRIPTION

This function returns **t** if *positive–integer* is a power of two, otherwise it returns **nil**.

REFERENCES

See also the related function **next–power–of–two–>=**.

The **power–of–two–p** function is most useful in combination with the following VP set definition operators:

def–vp–set **create–vp–set** **let–vp–set**

---

ppme [*Macro*]

"Pretty-print Macroexpand", used to examine code produced by the *Lisp compiler.

SYNTAX

ppme *form*

ARGUMENTS

form	*Lisp form to be macroexpanded and pretty-printed.

RETURNED VALUE

nil	Used for side effect only.

SIDE EFFECTS

Pretty-prints the macroexpansion (and thus the *Lisp compilation) of the *form.*

DESCRIPTION

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

The *Lisp compiler includes a macro that you can use to display the expanded form of a piece of code. Called **ppme** (short for "pretty print macroexpand"), it essentially performs a call to **pprint** and **macroexpand-1** to display the expanded form of a piece of *Lisp code.

EXAMPLES

A sample call to **ppme** is

```
> (setq *safety* 0)
> (ppme (*set (the single-float-pvar a)
              (+!!  (the single-float-pvar b)
                    (the single-float-pvar c))))
```

The resulting compiled code looks like this:

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

NOTES

Usage Note:

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

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

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

ppp [*Macro*]

Prints the values of the supplied pvar in neatly formatted style.

SYNTAX

ppp *pvar* **&key** **:mode :format :per–line :title :start :end :ordering :processor–list :print–arrays :return–argument–pvar :pretty :stream**

ARGUMENTS

pvar	Pvar expression. Pvar to be printed.
:mode	Either of **:cube** or **:grid**. Determines mode (send/grid) of formatted output. Defaults initially to **:cube**. (See Notes section, below.)
:format	String. Format directive used to print each value. Defaults initially to **"~s "**.
:per–line	Integer or **nil**. Number of values to print on the same line. Defaults initially to **nil**, indicating that no line-breaks are to be printed.
:title	String or **nil**. Text to display as title line, or **nil** for no title. Defaults initially to **nil**.
:start	Integer or list of integers. Send/grid address of processor at which to start formatting values. Defaults initially to **0**.
:end	Integer or list of integers. Send/grid address of processor at which to end formatting values. Default value is the current value of the global variable ***number–of–processors–limit***.
:ordering	List of integers or **nil**. Specifies order in which grid dimensions are traversed in formatting of values. This argument is meaningless unless the value of the **:mode** argument is **:grid**. Defaults initially to **nil**.

:processor-list — List of integers or **nil**. Send addresses of processors between **:start** and **:end** for which values are formatted. This argument is meaningless unless the value of the **:mode** argument is **:cube**. Defaults initially to **nil**.

:print-arrays — Scalar boolean. Determines whether arrays are displayed in full. Defaults to **t**.

:return-argument-pvar — Scalar boolean. Determines whether **ppp** returns the supplied *pvar* as its value. Defaults to **nil**.

:pretty — Scalar boolean. Value that Common Lisp global variable ***print-pretty*** is bound to during printing. Defaults to **nil**.

:stream — Stream object. Stream to which output is printed. Defaults to **nil**, which directs output to ***standard-output***. An argument of **t** directs output to ***terminal-io***.

RETURNED VALUE

pvar-or-nil — Depending on the value supplied for the **:return-argument-pvar** argument, either the supplied *pvar* argument or **nil**.

SIDE EFFECTS

Prints the selected values of *pvar* to the stream specified by the **:stream** argument.

DESCRIPTION

This macro is an alias for the macro **pretty-print-pvar**, which performs identically.

The **ppp** macro prints out the value of *pvar* in all specified processors, regardless of the currently selected set. If **ppp** accesses a processor that has no defined value for *pvar*, the output produced is not defined.

The keyword **:mode** can have the value **:cube** or **:grid**; in the latter case the pvar is printed out using grid addressing rather than cube addressing.

If the **:per-line** argument is **nil**, no newlines are ever printed between values; otherwise, the number of values specified by the **:per-line** argument are printed on each line.

The keyword **:format** has as its value a string that controls the printing format for each value; its value is used directly by the Common Lisp **format** function.

The **:ordering** keyword argument to **ppp** takes a list of integers specifying axes. It is valid only when used in conjunction with the **:grid** value of the **:mode** keyword and is most useful for printing a pvar defined in a VP set of more than two dimensions. With the **:ordering** keyword argument to **ppp**, the user can specify which "slices" of the *n*-dimensional grid are to be displayed. The last two dimensions specified in the **:ordering** list are the two dimensions that are shown as a single slice.

The keyword argument **:pretty** controls whether output values are pretty-printed. The value of the **:pretty** argument is bound as the value of the variable ***print-pretty*** for the duration of the call to **ppp**.

EXAMPLES

A sample call to **ppp** is

```
(ppp (self-address!!))
0 1 2 3 4 5 6 7 8 9 10 11 12 . . .
```

The output produced by **ppp** may be tailored by use of the many keywords. For example,

```
(ppp (self-address!!) :end 7)
0 1 2 3 4 5 6

(ppp (self-address!!) :start 6 :end 24
                      :per-line 6 :format "~3D")
  6  7  8  9 10 11
 12 13 14 15 16 17
 18 19 20 21 22 23

(ppp      (*!! (self-address!!) (self-address!!))
          :start 1 :end 4 :format "~R "
          :title "The monolith's dimensions are")

The monolith's dimensions are: one four nine
```

The **:processor-list** argument may be used to select specific processors to display, but only when the printing **:mode** is **:cube**, as it is by default. For example,

```
(ppp (*!! 2 (self-address!!))
     :processor-list '(1 2 3 5 7 11 13 17 19))
```

displays the output

```
2 4 6 10 14 22 26 34 38
```

The **:grid** option to the **:mode** keyword causes the output of **ppp** to be displayed in grid-address format. For example, assuming a two-dimensional grid,

```
(ppp (self-address!!) :mode :grid :end '(8 4) :format "~3D")
```

displays output similar to

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

The **:ordering** argument may be used to specify the order in which grid dimensions are displayed. For example,

```
(ppp (self-address!!) :mode :grid :end '(6 4)
                      :ordering '(1 0) :format "~3D")
```

displays output similar to

```
  0  4  8 12
  1  5  9 13
  2  6 10 14
  3  7 11 15
 16 20 24 28
 17 21 25 29
```

The keyword argument **:pretty** can be used to cause the output of some pvar values to be displayed in a cleaner format. Calling **ppp** on a structure pvar, for example, yields output such as the following:

```
#S(PERSON :NAME 0 :AGE 0 :SEX NIL) #S(PERSON :NAME 0 :AGE 0 :SEX
NIL) #S(PERSON :NAME 0 :AGE 0 :SEX NIL)
```

If the keyword argument **:pretty** is given the value **t**, this structure is printed as:

```
#S(PERSON :NAME 0
      :AGE 0
      :SEX NIL)
#S(PERSON :NAME 0
      :AGE 0
      :SEX NIL)
#S(PERSON :NAME 0
      :AGE 0
      :SEX NIL)
```

NOTES

There are global variables that specify the defaults for each of the keyword arguments except:

:pretty	**:print–arrays**	**:return–argument–pvar**
:stream		

See Chapter 2, "*Lisp Global Variables," in Part I of this Dictionary for a list of these variables and the default values to which they are initially bound.

Simulator Note:

The number of processors defined by default in the *Lisp simulator is very much lower than the number of processors generally available using CM hardware. Therefore, while using the *Lisp simulator, if **ppp** is called with no keyword arguments, as in

```
(ppp data-pvar)
```

then only a few values will be displayed. The same call to **ppp** executed with CM hardware attached can potentially display thousands or millions of values. When using CM hardware, it is prudent to use the **:start** and **:end** keywords (or the global variables controlling their defaults) to limit the number of values displayed.

REFERENCES

See also these related pvar pretty-printing operations:

ppp!!	
ppp–address–object	**ppp–css**
pppdbg	**ppp–struct**
pretty–print–pvar	**pretty–print–pvar–in–currently–selected–set**

ppp!! *[Macro]*

Prints the values of the supplied pvar in neatly formatted style, and returns the supplied pvar as its value.

SYNTAX

ppp!! *pvar* **&rest** *keyword–args*

ARGUMENTS

pvar	Pvar expression. Pvar to be printed and returned.
keyword–args	Keyword arguments. Accepts same keyword arguments as **ppp**.

RETURNED VALUE

pvar	The supplied *pvar* argument is returned.

SIDE EFFECTS

Prints selected values of *pvar* to the stream specified by the **:stream** argument.

DESCRIPTION

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**, with the exception of **:return–argument–pvar**.

NOTES

There are global variables that specify the defaults for each of the keyword arguments. See Chapter 2, "*Lisp Global Variables," in Part I of this Dictionary for a list of these variables.

REFERENCES

See also these related pvar pretty-printing operations:

ppp	
ppp–address–object	**ppp–css**
pppdbg	**ppp–struct**
pretty–print–pvar	**pretty–print–pvar–in–currently–selected–set**

ppp–address–object [*Function*]

Prints the values of the supplied address-object pvar in neatly formatted style.

SYNTAX

ppp–address–object *address–object–pvar* **&key :title :start :end :mode**

ARGUMENTS

address–object–pvar
Address-object pvar. Pvar to be printed.

:mode Either of **:cube** or **:grid**. Determines mode (send/grid) of formatted output.

:title String or **nil**. Text to display as title line, or **nil** for no title.

:start Integer. Send address of processor at which to start formatting values.

:end Integer. Send address of processor at which to end formatting values.

RETURNED VALUE

nil Evaluated for side effect only.

SIDE EFFECTS

Prints selected values of *address–object–pvar* to ***standard–output*** stream.

DESCRIPTION

This function is a specialized pretty-printer for address-object pvars.

NOTES

There are global variables that specify the defaults for each of the keyword arguments. See Chapter 2, "*Lisp Global Variables," in Part I of this Dictionary for a list of these variables.

REFERENCES

See also these related pvar pretty-printing operations:

ppp	**ppp!!**
ppp–css	
pppdbg	**ppp–struct**
pretty–print–pvar	**pretty–print–pvar–in–currently–selected–set**

ppp–css [*Macro*]

Prints out the send address, and the value of the supplied pvar, for each processor of the currently selected set

---

SYNTAX

ppp–css *pvar* **&key :format :start :end :title :mode**

ARGUMENTS

pvar	Pvar expression. Pvar from which values are printed.
:format	String. Format directive used to print each value.
:start	Integer. Send address of processor at which to start formatting values.
:end	Integer. Send address of processor at which to end formatting values.
:title	String or **nil**. Text to display as title line, or **nil** for no title.
:mode	Either of **:cube** or **:grid**. Determines mode (send/NEWS) of formatted output.

RETURNED VALUE

nil	Evaluated for side effect only.

SIDE EFFECTS

Prints send addresses and values from *pvar* to the ***standard–output*** stream.

DESCRIPTION

This macro is an alias for **pretty–print–pvar–in–currently–selected–set.**

NOTES

There are global variables that specify the defaults for each of the keyword arguments. See Chapter 2, "*Lisp Global Variables," in Part I of this Dictionary for a list of these variables.

REFERENCES

See also these related pvar pretty-printing operations:

ppp	**ppp!!**
ppp–address–object	
pppdbg	**ppp–struct**
pretty–print–pvar	**pretty–print–pvar–in–currently–selected–set**

pppdbg [*Macro*]

Prints the values of the supplied pvar in neatly formatted style, displaying the form that is evaluated to provide the pvar as a title.

SYNTAX

pppdbg *pvar* **&rest** *keyword–args*

ARGUMENTS

pvar	Pvar expression. Pvar to be printed.
keyword–args	Keyword arguments. Accepts same keyword arguments as **ppp**.

RETURNED VALUE

pvar–or–nil	Depending on the value supplied for the **:return–argument–pvar** argument, either the supplied *pvar* argument or **nil**.

SIDE EFFECTS

Prints selected values of *pvar* to the stream specified by the **:stream** argument.

DESCRIPTION

This macro is equivalent to **ppp**, except that the **:title** keyword argument defaults, not to **nil** (no title), but to the original form supplied as the *pvar* argument for **pppdbg**. The argument *pvar* may be any pvar. The *keyword–args* are identical to those for **ppp**.

EXAMPLES

For example, the expression

```
(pppdbg (self-address!!) :end 10)
```

displays the following:

```
(SELF-ADDRESS!!): 0 1 2 3 4 5 6 7 8 9
```

NOTES

There are global variables that specify the defaults for each of the keyword arguments. See Chapter 2, "*Lisp Global Variables," in Part I of this Dictionary for a list of these variables.

REFERENCES

See also these related pvar pretty-printing operations:

ppp	**ppp!!**
ppp-address-object	**ppp-css**
ppp-struct	
pretty-print-pvar	**pretty-print-pvar-in-currently-selected-set**

ppp–struct

[Function]

Prints the contents of the supplied structure pvar in a readable format.

SYNTAX

ppp–struct *pvar per–line* **&key :start :end :print–array**
:stream :width :title

ARGUMENTS

pvar	Structure pvar. Pvar to print in readable format.
per–line	Positive integer. Number of values to display per line.
:start	Integer. Send address of processor at which printing starts. Defaults to 0.
:end	Integer. Send address of processor at which to printing ends. Defaults to ***number–of–processors–limit***.
:print–array	Boolean. Determines whether arrays are printed out in full. Defaults to **t**.
:stream	Stream object or **t**. If supplied, output is written to the specified stream. Defaults to **t**, sending output to ***standard–output***.
:width	Integer. Width, in characters, of each value displayed. Defaults to 8 characters.
:title	String or **nil**. Text to display as title line, or **nil** for no title. Defaults to name of *pvar*'s structure type.

RETURNED VALUE

nil	Evaluated for side effect only.

SIDE EFFECTS

The contents of *pvar* from processor *start* up to processor *end* is written to *stream* in a readable format.

DESCRIPTION

The function **ppp–struct** attempts to print out the structure pvar *pvar* in readable format, with processor values for each slot being shown left to right, one line per slot. The number of values displayed per line is determined by *per–line*.

The keyword arguments **:start**, **:end**, **:print–array**, and **:stream** control the amount, format, and destination of the output exactly as with **ppp**.

The argument **:width** determines the printed width of each slot value, and defaults to 8 characters.

The argument **:title** defaults to **t**, which specifies that the title printed out is the name of the ***defstruct** of which *pvar* is an instance of. If **nil**, no title is printed out. If it is a string, then that string is used as the title.

EXAMPLES

```
(*defstruct person
  (ssn 0 :type (unsigned-byte 32))
  (age 0 :type (unsigned-byte 16))
  (height 0.0 :type single-float)
  (weight 0.0 :type single-float)
  )

  (ppp-struct a-person 5 :end 10 :width 10)

*DEFSTRUCT PERSON

SSN:     219101296 545417079 833166928 508389095 945762998
AGE:     43        76        9         96        63
HEIGHT:  0.7566829 6.0384245 6.8458276 2.9526687 6.9201202
WEIGHT:  52.873016 11.53174  29.510529 223.5896  244.65019

SSN:     604959766 822929695 445946453 856011938 684206262
AGE:     27        28        88        68        98
HEIGHT:  2.01059   5.2301087 6.1360407 1.8808416 6.9195743
WEIGHT:  82.76129  200.76877 165.2837  48.37853  154.92798

NIL
```

NOTES

There are global variables that specify the defaults for each of the keyword arguments except:

- **:print–array**
- **:stream**
- **:width**

See Chapter 2, "*Lisp Global Variables," in Part I of this Dictionary for a list of these variables.

REFERENCES

See also these related pvar pretty-printing operations:

ppp	**ppp!!**
ppp–address–object	**ppp–css**
pppdbg	
pretty–print–pvar	**pretty–print–pvar–in–currently–selected–set**

pref [*Macro*]

Retrieves the value of the supplied pvar in a single processor.

SYNTAX

pref *pvar–expression send–address* **&key :vp–set**

ARGUMENTS

pvar–expression	Pvar or pvar expression. Pvar from which value is accessed.
send–address	Integer or address object. Send address of processor from which value is accessed.
:vp–set	VP set object. VP set to which the result of *pvar–expression* belongs. Defaults to the value of ***current–vp–set***.

RETURNED VALUE

scalar–value	Value obtained by evaluating *pvar–expression* with the single processor specified by *send–address* selected.

SIDE EFFECTS

None.

DESCRIPTION

This macro returns, as a Lisp value, the value of *pvar–expression* in the processor specified by *send–address*. The pvar returned by *pvar–expression* may be any type of pvar, and may belong to any VP set.

The **:vp–set** argument determines the VP set in which the supplied *pvar–expression* is evaluated. If a **:vp–set** argument is not specified, *pvar–expression* is assumed to belong to the current VP set. It is only necessary to supply a value for the **:vp–set** argument if pvar-expression is an expression that must be evaluated in a VP set other than the current VP set.

EXAMPLES

The expression

```
(pref foo 17)
```

returns the value of pvar **foo** in processor 17.

The macro ***setf** may be applied to **pref** to store a value into a single processor of a pvar. For example, the expression

```
(*setf (pref foo 17) (* 19 99))
```

sets the value of pvar **foo** in processor 17 to 1881.

The *send–address* argument may reference *any* processor; it is not limited to processors in the currently selected set. The **pref** macro may be used to access any processor, whether or not that processor is currently active, in which the *pvar–expression* contains valid data.

For example, the result returned by the expression

```
(*all
  (*let ((x (self-address!!)))
    (*when (<!! (self-address!!) (!! 10))
      (pref x 30))))
```

is defined, even though the call to ***when** deselects processor 30. The contents of the local pvar **x** is set in all processors prior to the call to ***when**, so that when **pref** is called to access the value of **x** in processor 30, that value is defined.

The result of the following similar expression is not defined, however.

```
(*all
  (*when (<!! (self-address!!) (!! 10))
    (*let ((x (self-address!!)))
      (pref x 30))))
```

This example is in error, for the contents of **x** are determined after the currently selected set has been restricted, excluding processor 30. The local pvar **x** therefore has no defined value in that processor. The value returned by this example is undefined.

The **pref** function may be used to read values using grid addresses in either of two ways. One way is to call the function **cube–from–grid–address** (or **cube–from–vp–grid–address**), as in

```
(pref data-pvar (cube-from-grid-address 10 5 2 4))
```

(assuming that **data–pvar** belongs to a four-dimensional VP set). The other is to supply an address object by calling the function **grid**, as in

```
(pref data-pvar (grid 10 5 2 4))
```

NOTES

Performance Note:

To read a single array element from an array pvar there are two possibilities. The first is to copy the entire array containing the element from the CM to the front end, and then to reference the element itself. The second and much faster method is to perform a parallel array reference on the CM, and then to select a single value from the resulting pvar.

As a specific example, assume an array pvar has been defined by

```
(*defvar my-array-pvar
    (vector!! (self-address!!) (-!! (self-address!!))))

(pref my-array-pvar 3)
#(3 -3)
```

The first method copies an entire array from **my–array–pvar** with **pref**, and then uses the Common Lisp **aref** operator to reference a single array element on the front end. For example,

```
(aref (pref my-array-pvar 3) 1)
-3
```

The second method performs a parallel array reference on the CM with **aref!!**, and then uses **pref** to access a single value from the resulting pvar.

```
(pref (aref!! my-array-pvar (!! 1)) 3)
-3
```

This second method is much faster for array pvars containing large arrays because less data is transmitted between the CM and the front end. Even for expressions involving small arrays, the second method is more efficient because the *Lisp compiler is able to recogize and compile expressions of this type.

Of course, this same principle applies to reading data from a single slot of a structure pvar. It is in general more efficient to perform a parallel reference on the CM than it is to copy an entire array or structure from the CM to the front end and performing a serial reference on the front end.

Usage Note:

The global variable ***lisp-i:*pref-subselects-processors*** determines whether the pref operation evaluates its *pvar-expression* argument in all active processors, or whether evaluation takes place only in the processor specified by *send-address*.

If ***lisp-i:*pref-subselects-processors*** is set to **nil**, the default, then **pref** evaluates its *pvar-expression* argument in all active processors, regardless of the value of the *send-address* argument.

If ***lisp-i:*pref-subselects-processors*** is set to **t**, then **pref** evaluates its *pvar-expression* argument with only the single processor specified by *send-address* selected.

REFERENCES

See also the **!!** operator, which takes a single value and broadcasts it to all processors.

See also the following four operations that move more than one element at a time between the front end and the CM:

array-to-pvar	**array-to-pvar-grid**
pvar-to-array	**pvar-to-array-grid**

See also the related operations:

pref!!	***pset**	***set**	***setf**

pref!! [*Macro*]

Performs a parallel retrieval of values from the supplied pvar.

SYNTAX

pref!! *pvar–expression send–address–pvar* **&key :collision–mode :vp–set**

ARGUMENTS

pvar–expression	Pvar expression. Pvar from which values will be retrieved.
send–address–pvar	Pvar containing send addresses or address objects. Address of processor from which the value of *pvar–expression* is retrieved.
:collision–mode	A symbol. Must be one of **:collisions–allowed**, **:no–collisions**, **:many–collisions**, or **nil**. Specifies method used to resolve collisions. Defaults to **nil**.
:vp–set	VP set object. VP set to which the pvar returned by *pvar–expression* belongs. Defaults to VP set of *pvar–expression*. If *pvar–expression* is an expression rather than a pvar, this argument defaults to ***current–vp–set***.

RETURNED VALUE

pref–pvar	Temporary pvar. In each active processor, contains the value of *pvar–expression* in the processor whose address is the corresponding value of *send–address–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The **pref!!** macro is an interprocessor and inter-VP set communication operation. It returns a pvar containing in each active processor the value of *pvar–expression* in the processor specified by *send–address–pvar.*

Each active processor retrieves a value from the pvar returned by *pvar–expression.* Specifically, each processor retrieves the value of *pvar–expression* in the processor specified by the value of *send–address–pvar.*

The processors from which these values are being retrieved need not be in the currently selected set. Also, *pvar–expression* need not be in the current VP set. The **pref!!** operation allows data to be retrieved from non-active processors and from pvars in VP sets other than the current one.

The keyword argument **:collision–mode** determines the communication method used when there are collisions. A collision occurs when a single value of *pvar–expression* is accessed by more than one processor, i.e., when the value of *send–address–pvar* is the same in two or more active processors. The Connection Machine arranges that all processors involved in a collision get the same value, but depending on the number of collisions that occur, one of a number of strategies may be used to provide efficient communication.

The **:collision–mode** argument has four legal values:

- **:no–collisions**

  This option asserts that no two processors will attempt to reference the same value. If two processors do attempt to access the same value, the result is undefined. The **:no–collisions** option is significantly faster than any of the options that allow collisions, with the exception of the **nil** option.

- **:collisions–allowed**

  This option asserts that collisions are allowed, but that relatively few collisions will actually occur. The time required to complete the **pref!!** operation is proportional to the maximum number of processors involved in a collision.

- **:many–collisions**

  This option asserts that many collisions will occur, and is especially useful when large numbers of processors are accessing the same value. This option is slower than the preceding two, but the algorithm used ensures that the **pref!!** operation takes constant time regardless of the number of collisions.

- **nil**

  This option is the default, and asserts that any number of collisions may occur. While this option is faster than either **:collisions–allowed** or **:many–collisions,** and can even be faster than **:no–collisions** in some cases, it uses significantly more memory. If this option requires more memory than is currently available, the **:many–collisions** option will automatically be used instead.

The **:collision–mode** argument allows *Lisp to optimize communication in cases where each value of *send–address–pvar* is unique (i.e., **:no–collisions**), or when many values of *send–address–pvar* are the same (i.e., **:many–collisions**). Note that this argument represents an assertion by the user about what can be expected to happen. If this assertion is violated, the **pref!!** operation will run much more slowly. In the case of the **:no–collisions** option, some data can be lost, as well.

The **:vp–set** argument determines the VP set in which the supplied *pvar–expression* is evaluated. It is only necessary to supply a value for the **:vp–set** argument if *pvar–expression* is an expression that must be evaluated in a VP set other than the current VP set.

The *send–address–pvar* argument specifies the send addresses of processors either in the current VP set or another VP set. If *pvar–expression* is a symbol bound to a pvar and no **:vp–set** argument is specified, the values of *send–address–pvar* are interpreted relative to the VP set to which *pvar–expression* belongs. If *pvar–expression* is an expression and no **:vp–set** argument is specified, the values of *send–address–pvar* are interpreted relative to the ***current–vp–set***. If *pvar–expression* is an expression and a **:vp–set** argument is specifed, the values of *send–address–pvar* are interpreted relative to the **:vp–set** argument.

The actual evaluation of *pvar–expression* is performed only in those processors from which values are being retrieved. Both the *send–address–pvar* and **:vp–set** arguments are used to determine the set of processors in which *pvar–expression* is evaluated:

- If *pvar–expression* is a symbol bound to a pvar, then *pvar–expression* is evaluated in the set of processors specified by *send–address–pvar* in the VP set to which *pvar–expression* belongs.

- If *pvar–expression* is an expression and no **:vp–set** argument is provided, then *pvar–expression* is evaluated in the set of processors specified by *send–address–pvar* in the current VP set.

- If *pvar–expression* is an expression and a **:vp–set** argument is specified, then *pvar–expression* is evaluated in the set of processors specified by *send–address–pvar* in the VP set specified by the **:vp–set** argument.

Examples of these three cases are shown below.

EXAMPLES

Here is a sample call to **pref!!**:

```
(*defvar pvar-a (random!! (!! 10)))
(*defvar pvar-b)
(*set pvar-b (pref!! pvar-a (self-address!!)))
```

The value of **pvar–a** in each processor is copied and returned by **pref!!**, and stored in **pvar–b** by ***set**. In this example, no interprocessor communication takes place; each processor is simply getting data from itself.

More interesting uses of **pref!!** involve exchanging values between processors. For example, the expression

```
(*set backwards-pvar
    (pref!! pvar (-!!   (!! (1- *number-of-processors-limit*))
                        (self-address!!))))
```

stores the values of **pvar** into **backwards–pvar** in reverse order of send addresses.

The expression

```
(*set pvar-a
    (pref!! pvar-a (mod!! (1-!! (self-address!!))
                          (!! *number-of-processors-limit*))))
```

shifts the value of **pvar–a** in each processor to the processor with the next higher send address (with wraparound).

This example demonstrates that *pvar–expression* is evaluated only in the processors from which values are being retrieved:

```
(*all
    (*when (not!! (=!!  (self-address!!)
                        (!! (1- *number-of-processors-limit*))))
      (ppp (pref!! (/!! (!! 1.0) (self-address!!))
                   (1+!! (self-address!!)))
           :end 4)))

1.0 0.5 0.3333334 0.25
```

Each processor retrieves data from its successor in send address order. (The call to ***when** excludes the processor with the highest address.) If the expression

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

was evaluated in the currently selected set of processors, including processor 0, then a division by zero would occur. However, no processor retrieves a value from processor 0 in the above example, so processor 0 does not evaluate the division form in the call to **pref!!**, and no division by zero occurs. Note also that a value is retrieved from the processor with the highest address, even though that processor is not currently active.

The next example demonstrates that *pvar–expression* is evaluated in the VP set specified by the **:vp–set** argument, and only in the processors in that VP set from which values are retrieved (in this case a single processor).

```
(def-vp-set fred '(256 256))
(*defvar fred-pvar (self-address-grid!! (!! 0))
        "Fred X coordinate" fred)
(def-vp-set barney '(65536))

(*with-vp-set barney
  (ppp
    (pref!!
      (progn
           (format t "~%The current vp set is ~S"
                *current-vp-set*)
      (format t "~%The number of active processors is ~S"
                (*sum (!! 1)))
        fred-pvar)
      (grid!! (!! 25) (!! 25))
      :vp-set fred)
    :end 5))
```

This example produces the following output:

```
The current vp set is #<VP-SET Name: FRED, Dimensions . . . >
The number of active processors is 1
25 25 25 25 25
```

The **pref!!** operation can also be used to transfer values between different VP sets, as in the following example.

```
(*proclaim '(type (pvar (unsigned-byte 4))
                  matrix diagonal-elements))

(def-vp-set diagonal-vp-set '(8192)
    :*defvars ((diagonal-elements (!! 0))))

(def-vp-set matrix-vp-set '(128 128)
    :*defvars ((matrix (random!! (!! 10)))))
```

These forms define two VP sets, **diagonal–vp–set** and **matrix–vp–set**, with one and two dimensions respectively. Two pvars are also defined, one associated with each VP set, that have the following initial values:

```
(ppp matrix :mode :grid :end '(5 5))

  DIMENSION 0 (X)  ----->

5 6 3 5 6
4 9 4 5 6
3 9 1 5 2
2 6 2 3 9
4 0 9 3 4

(ppp diagonal-elements :end 5)

0 0 0 0 0
```

The following function uses **pref!!** to copy values from **matrix** that are stored along the diagonal of the **matrix–vp–set** grid into the **diagonal–elements** pvar.

```
(defun retrieve-diagonal-elements ()
  (*with-vp-set diagonal-vp-set  ;;; VP set of dest-pvar
    (*when (<!! (self-address!!) (!! 128))
      (*set diagonal-elements
            (pref!! matrix-vp-set
                    (cube-from-vp-grid-address!!
                      ;; Treat pair of send addresses from one-d
                      ;; as grid address in two-d, and convert
                      ;; to corresponding send address in two-d
                      (self-address!!)
                      (self-address!!))
                :vp-set matrix-vp-set     ; VP set of dest-pvar
                    )))))
```

Following a call to **retrieve–diagonal–elements**, the **matrix** and **diagonal–elements** pvars display as:

```
(ppp matrix :mode :grid :end '(5 5))

  DIMENSION 0 (X)  ----->

5 6 3 5 6
4 9 4 5 6
3 9 1 5 2
2 6 2 3 9
4 0 9 3 4

(ppp diagonal-elements :end 5)

5 9 1 3 4
```

Note the use of **cube–from–vp–grid–address!!** to determine the send addresses of the diagonal elements in **matrix–vp–set**. The send address of each element of the **diagonal–elements** pvar is used twice to form a grid address along the diagonal of the **matrix** pvar. This grid address is then converted by **cube–from–vp–grid–address!!** into the appropriate send address within **matrix–vp–set**.

Another way of converting grid addresses to send addresses within a **pref!!** form is the use the **grid!!** function. For instance, the above call to **pref!!** could have been written as

```
(pref!! matrix (grid!! (self-address!!) (self-address!!)))
```

See the definition of **grid!!**, and Section 6.5, "Address Objects" of the **Lisp Reference Supplement*, Version 5.0, for more information.

NOTES

Usage Note:

The default value (**nil**) of **:collision–mode** invokes the 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 can require much more temporary memory.

Performance Note:

A call to **pref!!** with no collisions is implemented using two calls to ***pset**: one to send the address of the processor requesting the data to the processor from which the data is to be retrieved, and another to send the data requested back to the requesting processor.

It is often possible to rewrite an algorithm that uses **pref!!** (in which data is retrieved) into an algorithm using ***pset** (in which data is sent, rather than retrieved), halving the communications time required.

For example

```
(*when (<!! (self-address!!) (!! 100))
  (*set dest (pref!! source
                     (+!! (self-address!!) (!! 100)))))
```

could be rewritten as

```
(*when (and!! (<=!! (!! 100) (self-address!!))
              (<!! (self-address!!) (!! 200)))
  (*pset source dest (-!! (self-address!!) (!! 100))))
```

Style Note:

The **pref!!** macro may be used with ***setf**. However, a call to ***setf** of the form

```
(*setf (pref!! dest-pvar address-pvar) source-pvar)
```

is equivalent to a call to ***pset** of the form

```
(*pset :no-collisions source-pvar dest-pvar address-pvar)
```

Calling ***pset** directly in this case is preferable as being more readable.

REFERENCES

See also the macro ***pset**, which performs a parallel store operation.

See also these related NEWS communication operators:

***news** **news!!** **news–border!!**
***news–direction** **news–direction!!**

See also these related off-grid processor address tests:

off–grid–border–p!! **off–grid–border–relative–direction–p!!**
off–grid–border–relative–p!! **off–vp–grid–border–p!!**

pretty–print–pvar [*Macro*]

Prints the values of the supplied pvar in neatly formatted style.

SYNTAX

pretty–print–pvar *pvar* **&key :mode :format :per–line :title :start :end :ordering :processor–list :print–arrays :return–argument–pvar :pretty :stream**

ARGUMENTS

See the entry for **ppp** for a description of the arguments.

RETURNED VALUE

pvar–or–nil Depending on the value supplied for the **:return–argument–pvar** argument, either the supplied *pvar* argument or **nil**.

SIDE EFFECTS

Prints the selected values of *pvar* to the stream specified by the **:stream** argument.

DESCRIPTION

This macro has an alias **ppp**, which operates identically. See the definition of **ppp** for information and examples of both of these macros.

REFERENCES

See also these related pvar pretty-printing operations:

ppp!! **ppp–address–object** **ppp–css**
pppdbg **ppp–struct**
pretty–print–pvar–in–currently–selected–set

pretty–print–pvar–in–currently–selected–set [*Macro*]

Prints out the send address and value of the supplied pvar for all processors in the currently selected set.

SYNTAX

pretty–print–pvar–in–currently–selected–set *pvar*
&key :format :start :end :title :mode

ARGUMENTS

pvar	Pvar expression. Pvar from which values are printed.
:format	String. Format directive used to print each value.
:start	Integer. Send address of processor at which to start formatting values.
:end	Integer. Send address of processor at which to end formatting values.
:title	String or **nil**. Text to display as title line, or **nil** for no title.
:mode	Either of **:cube** or **:grid**. Determines mode (send/grid) of formatted output.

RETURNED VALUE

nil	Evaluated for side effect only.

SIDE EFFECTS

Prints send addresses and values from *pvar* to the ***standard–output*** stream.

DESCRIPTION

This function prints out the the cube address and value of *pvar* for all processors in the currently selected set.

NOTES

There are global defaults for each of the keyword arguments.

See Chapter 2, "*Lisp Global Variables," for a list of these variables.

REFERENCES

This macro has an alias, **ppp–css**.

See also these related pvar pretty-printing operations:

ppp	**ppp!!**
ppp–address–object	
pppdbg	**ppp–struct**
pretty–print–pvar	

*processorwise [*Defun]

Converts a sideways (slicewise) array to the normal, processorwise orientation.

SYNTAX

***processorwise** *array–pvar*

ARGUMENTS

array–pvar	Array pvar. Sideways (slicewise) array pvar to be converted.

RETURNED VALUE

t	Evaluated for side effect only.

SIDE EFFECTS

Converts *array–pvar* from sideways to normal, processorwise orientation.

DESCRIPTION

Converts a sideways (slicewise) array to a normal, processorwise orientation.

The *array–pvar* parameter must be a sideways (slicewise) array, otherwise an error is signaled.

NOTES

The function ***processorwise** is equivalent to a call to ***sideways–array** with an array argument that is in sideways (slicewise) orientation.

There are some important restrictions on the size of arrays passed as arguments to ***processorwise**.

The *array–pvar* argument must be an array pvar that contains elements whose lengths are powers of 2 or multiples of 32. Further, the total number of bits the array occupies in CM memory must be divisible by 32. This number can be determined either by **(pvar-length array-pvar)** or by multiplying the total number of elements in the array by the size of an individual element.

The ***processorwise** function is most efficient when the array elements of *array-pvar* are each 32 bits long.

REFERENCES

See also the functions ***sideways-array, sideways-array-p**, and ***slicewise.**

*proclaim [*Macro*]

Records a global declaration about *Lisp variables and functions. Also provides the *Lisp compiler with information about Common Lisp variables.

SYNTAX

***proclaim** *declaration*

ARGUMENTS

declaration	*Lisp declaration form. Proclamation to be recorded. This argument is evaluated, so declaration forms must be quoted.

RETURNED VALUE

nil	Evaluated for side effect only.

SIDE EFFECTS

Records *declaration* as a global declaration about *Lisp variables and functions.

DESCRIPTION

The *Lisp version of the Common Lisp **proclaim** function. It is used to make global declarations, including the data types of global pvar variables and user-defined functions.

EXAMPLES

The ***proclaim** macro is commonly used in five ways:

- To provide type declarations for permanent pvars defined by ***defvar**.

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

(*proclaim
 '(type (vector-pvar (array (unsigned-byte 32) (4 4)) 3)
     my-nested-array1 my-nested-array2))
(*defvar my-nested-array1)
(*defvar my-nested-array2)
```

- To provide function declarations so that the *Lisp Compiler has information regarding the returned value of user-defined *Lisp functions.

  For example,

```
(*proclaim
 '(ftype (function (single-float-pvar single-float-pvar)
                   single-float-pvar)
         hypotenuse!!))
```

  informs the *Lisp compiler that the **hypotenuse!!** function takes two single float pvars as arguments and returns a single float pvar as a result.

  The expression

```
(*proclaim '(ftype (function (&rest t) (pvar boolean))
                    my-and!!))
```

  informs the *Lisp compiler that the **my-and!!** function takes any number of arguments of any type, and returns a boolean pvar.

  Currently, the *Lisp compiler does not use the information about arguments provided in **function** or **ftype *proclaim** forms. The declaration for each argument in these forms may be completely specified for documentation purposes, or may be specified simply as **t**. However, the number of argument declarations provided must match the number of arguments accepted by the function.

- To provide the *Lisp compiler with information about scalar variables used in pvar expressions. Note that ***proclaim** is used instead of **proclaim**, so that the *Lisp compiler will have access to the declarations.

```
(*proclaim '(type double-float two-pi))
(defparameter two-pi (* pi 2.0))

(*proclaim '(type fixnum x-dimension y-dimension))
```

```
(defvar x-dimension 3)
(defvar y-dimension 4)
```

- To define or change the compiler settings for the *Lisp compiler.

  For example,

  ```
  (*proclaim '(*optimize (safety 3)))
  ```

  informs the *Lisp compiler that full safety should be enabled globally. For more information about the *Lisp compiler and the many compiler settings available, see the **Lisp Compiler Guide*, Version 5.0.

- To inform the Lisp compiler that a symbol will later be defined with ***defun**, and will therefore be a macro rather than a function.

  For example,

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

  (defun bar (x) (foo x))

  (*defun foo (x) (*sum x))
  ```

  Without the call to ***proclaim**, when **bar** is compiled the call to **foo** is treated as a function call. When **foo** is defined with ***defun**, it is actually defined as a macro, so that the call to **foo** within **bar** will not execute properly. Declaring that **foo** will be defined by ***defun** prior to the definition of any function that calls **foo** allows Lisp to compile these functions properly.

NOTES

Syntax Notes:

The *declaration* argument of ***proclaim** must be quoted to prevent evaluation, just as in Common Lisp the declaration argument to proclaim must be quoted.

Also, nearly all calls to ***proclaim** end with a double parentheses, as the above examples show. It is a good rule of thumb to recheck any ***proclaim** form ending with a single parenthesis or with more than two parentheses, for it may contain an error. Note the exception given by the fourth example above. The use of ***proclaim** to declare the *Lisp compiler safety level ends in three parentheses, but is nevertheless correct.

Compiler Note:

The use of the Common Lisp **proclaim** operator to inform the *Lisp compiler of type information is obsolete and no longer supported.

REFERENCES

See also the related *Lisp declaration operators:

***locally** **unproclaim**

See also the related type translation function **taken-as!!**.

See also the related type coercion function **coerce!!**.

*pset [*Macro*]

Copies values from the source pvar into the destination pvar. This operation may be used to transfer values between processors in the same VP set and between processors in different VP sets.

SYNTAX

***pset** *combine–method source–pvar destination–pvar dest–address–pvar*
&key :notify :vp–set :collision–mode :combine–with–dest

ARGUMENTS

combine–method Keyword. Specifies the method used to combine multiple values sent to the same processor. Must be one of: **:default, :no–collisions, :overwrite, :or, :and, :logior, :logand, :add, :max, :min, :queue**

source–pvar Pvar from which values are copied. The value in each active processor must be of a data type that can legally be stored in *destination–pvar*. The *source–pvar* argument must belong to the current VP set.

destination–pvar Pvar into which values are copied. May belong to any VP set.

dest–address–pvar Pvar containing send addresses or address objects. Addresses must be valid for the VP set to which *destination–pvar* belongs.

:notify Boolean pvar used to indicate in which processors *destination–pvar* is altered. If supplied, it is set to **t** in those processors that receive a value. Must belong to the same VP set as *destination–pvar*.

:vp–set VP set object. If supplied, it must be the VP set to which *destination–pvar* belongs. Used for optimization purposes only.

:collision–mode The **:collision–mode** keyword argument is superfluous, and is retained for compatibility purposes.

:combine–with–dest Boolean. Controls whether or not the values already contained in the destination pvar are combined with the values being sent from the source processors. Defaults to **nil**, which causes the values of the destination pvar to be overwritten by values from the source pvar.

RETURNED VALUE

nil Evaluated for side effect only.

SIDE EFFECTS

In each processor specified by *dest–address–pvar*, *destination–pvar* is overwritten with either a single *source–pvar* value or a combination of *source–pvar* values.

If *notify–pvar* is supplied, it is set to **t** in each processor in which *destination–pvar* received a value; elsewhere it is unaffected.

DESCRIPTION

The ***pset** macro is an interprocessor and inter-VP set communication operation. It copies values from one pvar to another. Source values from one processor may be copied to a different processor. Also, *source–pvar* and *destination–pvar* may belong to different VP sets.

Using a mailbox analogy, the values in *source–pvar* are messages, the values in *dest–address–pvar* are the addresses of the mailboxes to which they are sent, and *destination–pvar* is the set of mailboxes into which the messages are delivered.

The arguments *value–pvar* and *dest–address–pvar* are only evaluated by the active processors of the current VP set. These arguments must be pvars belonging to the current VP set.

The *dest–pvar* argument may be any pvar in any VP set; it does not need to belong to the current VP set.

The *dest–address–pvar* may contain integer values that constitute valid send addresses for the VP set to which *dest–pvar* belongs. Alternatively, an address object pvar may be used as the value of the *dest–address–pvar* argument.

For all processors in the currently selected set, the value of *value–pvar* is sent to the processor addressed by *dest–address–pvar*, and stored into *destination–pvar* in the processor addressed by *dest–address–pvar*.

When *dest–address–pvar* contains duplicate addresses, some processors receive more than one value. When this occurs, the values received are combined according to the method specified by *combine–method*. The effect of each legal *combine–method* value is described below.

:default	An error is signaled if any processor receives more than one value.
:no–collisions	Asserts that no more than one value will be sent to each processor. If any processor does receive multiple values, the code is in error.
:overwrite	One arbitrarily selected value is stored; all other values are ignored.
:or	The logical OR is stored.
:and	The logical AND is stored.
:logior	The bitwise OR is stored. The *source–pvar* must contain integers only.
:logand	The bitwise AND is stored. The *source–pvar* must contain integers only.
:add	The numerical sum is stored.
:max	The numerical maximum is stored.
:min	The numerical minimum is stored.
:queue	Queues colliding values as a vector in the destination pvar.

The optional **:notify** argument must be a pvar. When ***pset** has finished executing, the value of the **:notify** pvar is **t** in each processor where *destination–pvar* has been altered, in other words, wherever a processor has received and stored a *source–pvar* value in *destination–pvar* — even if the value stored happens to be the same as the original value — the **:notify** pvar is set. The value of the **:notify** pvar is left unchanged in processors where the *destination–pvar* has not been altered.

If supplied, the *vp–set* argument must be the VP set to which *destination–pvar* belongs. This argument is available solely for optimization and readability. If a *vp–set* argument is not supplied, *Lisp determines the proper VP set from *destination–pvar*.

The *collision–mode* argument is superfluous as of Version 5.0, and is retained for compatibility purposes.

The **:combine–with–dest** argument controls whether or not the values already contained in the destination pvar are combined with the values being sent from the source processors. Defaults to **nil**, which causes the values of the destination pvar to be overwritten by values from the source pvar.

EXAMPLES

Here is a simple call to ***pset**:

```
(*defvar pvar-a (random!! (!! 10)))
(*defvar pvar-b)
(*pset :no-collisions pvar-a pvar-b (self-address!!))
```

The value of **pvar-a** in each processor is stored in the corresponding processor of **pvar-b**. Because there is no possibility of more than one value being sent to the same processor, the **:no-collisions** option is used to increase efficiency. This example is identical in operation to a call to ***set**:

```
(*set pvar-a pvar-b)
```

In this example, data is copied from one pvar to another within each processor, so no interprocessor communication takes place.

More interesting uses of ***pset** involve exchanging values between processors:

```
(defun backwards (pvar)
   (*let (backwards-pvar)
      (*pset :default pvar backwards-pvar
             (-!! (!! (1- *number-of-processors-limit*))
                  (self-address!!)))
       backwards-pvar))
```

This function takes any pvar and returns a copy of that pvar with its values in reverse send-address order. The ***pset** macro is used to transfer the value of **pvar** from each processor to the processor's opposite in terms of send addresses, where the value is stored in **backwards-pvar**. So, for example,

```
(*cold-boot :initial-dimensions '(10))
(ppp dest :end 10)
9 8 7 6 5 4 3 2 1 0
```

The next example is another function that calls ***pset**, this time to obtain the sum of the values of a pvar:

```
(defun my-*sum (pvar)
    (declare (type (pvar (unsigned-byte 10)) pvar))
    (pref (*let (the-sum-goes-here)
            (declare (type (pvar (unsigned-byte 32))
                         the-sum-goes-here))
            (*all (*pset :add pvar the-sum-goes-here (!! 47)))
            the-sum-goes-here)
          47))
```

The function **my-*sum** uses ***pset** to sum a pvar over all the Connection Machine processors. Each processor sends its value to the same address, processor 47 (any legal send address can be substituted for 47). The values are collected using the **:add** method, which calculates and stores the sum. The **pref** operation is then used to read and return the sum. (Note: the *Lisp function ***sum** performs the same operation much more efficiently than this example.)

An example of a realistic use for ***pset** is:

```
(*defvar data-pvar (random!! (!! 10)))

(defun histogram (pvar)
  (declare (type (pvar (unsigned-byte 4)) pvar))
  (*let ((histogram (!! 0)))
    (declare  (type (pvar (unsigned-byte
                            *current-send-address-length*))
                    histogram))
    (*pset :add (!! 1) histogram pvar)
    histogram))
```

This function creates and returns a histogram of the values in **pvar**. The call to ***pset** causes each processor to treat its value of **pvar** as a send address and send the value 1 to the processor at that address. The **:add** combine method is used, so each processor stores in **histogram** a count of the number of values in **pvar** which are the same as its send address. For example:

```
(*defvar data-pvar (random!! (!! 10)))
(ppp data-pvar :end 20)
5 3 9 4 1 7 0 9 1 4 1 9 2 0 9 0 3 6 0 7

(ppp (histogram data-pvar) :end 14)
5273 6397 6808 7468 6952 8403 7691 4569 7774 4201 0 0 0 0
```

This shows that, for example, there are 6808 occurrences of the value 2 in **data-pvar**.

The ***pset** macro may also be used to transfer values between VP sets, as in the following example.

```
(*proclaim '(type (pvar (unsigned-byte 16))
                  one-d-pvar two-d-pvar))

(def-vp-set one-d '(128)
    :*defvars ((one-d-pvar (1+!! (self-address!!)))))

(def-vp-set two-d '(128 128)
    :*defvars ((two-d-pvar (!! 0))))
```

These forms define two VP sets, **one–d** and **two–d**, with one and two dimensions respectively. The VP set **two–d** is defined as a square grid with as many processors along its edge as there are processors in **one–d**.

Two pvars are also defined, one associated with each VP set, having the following initial values:

```
(ppp one-d-pvar :end 10)
1 2 3 4 5 6 7 8 9 10

(ppp two-d-pvar :mode :grid :end '(5 5))

  DIMENSION 0 (X)  ----->

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

The following expression uses the ***pset** macro to copy **one–d–pvar** into **two–d–pvar** in such a way that the values of **one–d–pvar** are stored on the diagonal of the grid of the **two–d** VP set.

```
(*with-vp-set one-d   ;;; VP set of source-pvar
    (*pset :no-collisions one-d-pvar two-d-pvar
         (cube-from-vp-grid-address!! two-d
              ;;; Treat pair of send addresses from one-d
              ;;; as grid address in two-d, and convert
              ;;; to corresponding send address in two-d
              (self-address!!)
              (self-address!!))
         :vp-set two-d))  ;;; VP set of dest-pvar

(ppp two-d-pvar :mode :grid :end '(5 5))

  DIMENSION 0 (X)  ----->

1 0 0 0 0
0 2 0 0 0
0 0 3 0 0
0 0 0 4 0
0 0 0 0 5
```

Note the use of **cube–from–vp–grid–address!!** to convert send addresses from **one–d** into send addresses for **two–d** along the diagonal of the grid. The send address of each value of **one–d–pvar** is used twice to form a grid address along the diagonal of **two–d–pvar**. This grid address is then converted by **cube–from–vp–grid–address!!** to the appropriate send address within the **two–d** VP set.

Another way of converting grid addresses to send addresses within a ***pset** form is the use the **grid!!** function. For instance, the above call to ***pset** could have been written as

```
(*pset :no-collisions one-d-pvar two-d-pvar
       (grid!! (self-address!!) (self-address!!))
       :vp-set two-d)
```

See the definition of **grid!!**, and Section 6.5, "Address Objects" of the **Lisp Reference Supplement*, Version 5.0, for more information.

When **:combine-with-dest** is **nil** (the default), the source values and dest values are not combined, with the result that source values simply overwrite destination values in each processor. When **:combine-with-dest** is **t**, the source and dest values are summed.

The following function demonstrates this feature:

```
(defun show-combine-with-dest ()
 (*let (source dest)
   (declare (type (field-pvar 32) source dest))
   (*set source (self-address!!))
   (*set dest (self-address!!))
   (*pset :add source dest (self-address!!)
          :combine-with-dest nil)
   (ppp dest :end 4)
   (*set dest (self-address!!))
   (*pset :add source dest (self-address!!)
          :combine-with-dest t)
   (ppp dest :end 4)))
```

A sample call to this function looks like:

```
(show-combine-with-dest)
0 1 2 3
0 2 4 6
```

Finally, the following function definition shows how the **:notify** argument to ***pset** can be used:

```
(defun send-and-add (source dest address)
  "This function sums source into dest, and then counts"
  "How many processors actually summed up data."
  (*let (notify-pvar)
    (declare (type boolean-pvar notify-pvar))
    (*all (*set notify-pvar nil!!))
    (*pset :add source dest address :notify notify-pvar)
    (*all (*when notify-pvar
             (format t "~%~D processors summed data"
                     (*sum (!! 1)))))))
```

This function may be called with any number of processors selected. All processors are made active temporarily to initialize **notify-pvar**, and then a call is made to ***pset** to perform a send operation. The value of **notify-pvar** is then used to display the number of processors that actually transmitted data. First all processors are selected (since some processors receiving data may not be in the currently selected set), and then **notify-pvar** is used to select those processors that in fact received data. With these processors active, a call to ***sum** is made to return a count of those processors.

The **:queue** combiner specifies that *Lisp should use the Paris **cm:send-to-queue32-1l** instruction, which queues multiple values arriving at a single destination processor into an array. The first element of the array stores the number of values that have arrived at that processor.

The simplest way to think of using the **:queue** combiner is as a queue-structure ***defstruct**, such as the following:

```
(defparameter float-queue-length 6)

(*defstruct float-queue
    (count    0    :type (unsigned-byte 32))
    (vector   (make-array 6 :element-type 'single-float)
                       :type (vector single-float 6)))
(*proclaim '(type float-queue-pvar queue))

(*defvar queue)
```

A simple function that initializes this queue structure and uses the **:queue** combiner is:

```
(defun queue-example ()
    (*setf (float-queue-count!! queue) (!! 0))
    (*setf (float-queue-vector!! queue)
         (make-array!! 6:initial-element (!! -1.0)
                       :element-type 'single-float-pvar))
    (*when (<!! (self-address!!) (!! 6))
      (compiler-let ((*compilep* nil))
         (*pset :queue  (float!! (self-address!!)) queue
              (random!! (!! 6)))))
    (ppp queue :end 6))
```

Note that the *Lisp compiler does not recognize the **:queue** argument in Version 6.0, and thus the compiler must be disabled around the ***pset** form to prevent warning messages from being generated.

The output from a call to this function might be:

```
(queue-example)
#S(FLOAT-QUEUE :COUNT 1 :VECTOR #(2.0 0.0 0.0 0.0 0.0 0.0))
#S(FLOAT-QUEUE :COUNT 2 :VECTOR #(0.0 5.0 0.0 0.0 0.0 0.0))
#S(FLOAT-QUEUE :COUNT 1 :VECTOR #(1.0 0.0 0.0 0.0 0.0 0.0))
```

```
#S(FLOAT-QUEUE :COUNT 0 :VECTOR #(0.0 0.0 0.0 0.0 0.0 0.0))
#S(FLOAT-QUEUE :COUNT 0 :VECTOR #(0.0 0.0 0.0 0.0 0.0 0.0))
#S(FLOAT-QUEUE :COUNT 2 :VECTOR #(4.0 3.0 0.0 0.0 0.0 0.0))
```

If more values are received in a destination processor than can be stored in the array, arbitrary values in excess will be discarded. In this case the count value will reflect the total number of values received, regardless of whether they were discarded or not.

The **:queue** combiner has the restriction that the *destination–pvar* argument must have a length of at least 64 bits; 32 bits for the count, and 32 bits for at least one element. The length must also be a multiple of 32 bits. The *source–pvar* argument must be representable in 32 bits.

NOTES

The ***pset** macro invokes the general routing hardware of the Connection Machine. While providing flexibility in communication of values between processors, the general router is less efficient than the communication methods employed by more specialized operators, such as ***news, news!!** and **scan!!**.

Performance Considerations:

The **:or** and **:and** combination methods are faster if the *source–pvar* contains only boolean values (**t** or **nil**).

Cautions:

The **:notify** pvar argument is unaltered in processors where *destination–pvar* is unaltered. The implications are:

- This allows one to track the cumulative effects of multiple ***pset** calls.
- User code is responsible for the initial value of the **:notify** pvar. In many cases it is advisable to ***set** the **:notify** pvar to **nil!!** in all processors prior to executing ***pset**.

Errors:

It is an error if any value copied is of a data type that cannot be stored in *destination–pvar*.

It is an error if *source–pvar* and *destination–pvar* are structure pvars of a type defined to include a variable-length field, and if the length of that field is different in *source–pvar* and *destination–pvar*. For instance, if the length of the field is dependent on the value of ***current–send–address–length***, and if *source–pvar* and *destination–pvar* belong to VP sets of different sizes, then ***pset** will fail.

REFERENCES

The function ***pset** copies data from one pvar to another, much as ***set** does. However, ***pset** is also able to exchange data between processors, whereas ***set** performs only a straight copy operation. See the ***set** Dictionary entry for details.

See also the related processor communication operator **pref!!**.

See also these related NEWS communication operators:

***news**	**news!!**	**news–border!!**
***news–direction**	**news–direction!!**	

See also these related off-grid processor address tests:

off–grid–border–p!!	**off–grid–border–relative–direction–p!!**
off–grid–border–relative–p!!	**off–vp–grid–border–p!!**

pvar–exponent–length *[Function]*

Returns bit length of exponent of the supplied floating-point or complex pvars.

SYNTAX

pvar–exponent–length *pvar*

ARGUMENTS

pvar	Floating-point or complex pvar. Pvar for which exponent bit length is determined.

RETURNED VALUE

exponent–length	Integer. Length in bits of exponent field of supplied pvar.

SIDE EFFECTS

None.

DESCRIPTION

This function returns the bit-length of the exponent of *pvar*. The argument *pvar* may be any pvar, but only floating-point or complex pvars return meaningful values.

Note: This function has no meaning in the *Lisp simulator, and returns no useful value.

REFERENCES

See also the following general pvar information operators:

allocated–pvar–p	**describe–pvar**	
pvar–length	**pvar–location**	**pvar–mantissa–length**
pvar–name	**pvarp**	**pvar–plist**
pvar–type	**pvar–vp–set**	

---

pvar–length [*Function*]

Returns bit length of the CM field associated with the supplied *pvar.*

SYNTAX

pvar–length *pvar*

ARGUMENTS

pvar	Pvar expression. Pvar for which field bit length is determined.

RETURNED VALUE

bit–length	Integer. Length in bits of CM field associated with *pvar*.

SIDE EFFECTS

None.

DESCRIPTION

This function returns the bit length of the field associated with *pvar*. This function can be used to supply field length arguments in calls to Paris routines. The argument *pvar* may be any pvar.

Note: This function has no meaning in the *Lisp simulator, and returns no useful value.

REFERENCES

See also the following general pvar information operators:

allocated–pvar–p	**describe–pvar**	**pvar–exponent–length**
pvar–location	**pvar–mantissa–length**	
pvar–name	**pvarp**	**pvar–plist**
pvar–type	**pvar–vp–set**	

pvar–location *[Function]*

Returns field-id of the CM field associated with the supplied *pvar*.

SYNTAX

pvar–location *pvar*

ARGUMENTS

pvar Pvar expression. Pvar for which field-id is determined.

RETURNED VALUE

field–id Integer. Field-id of CM field associated with *pvar*.

SIDE EFFECTS

None.

DESCRIPTION

This function returns the field-id of the field associated with *pvar*. This function can be used to supply field-id arguments in calls to Paris routines. The argument *pvar* may be any pvar.

Note: This function has no meaning in the *Lisp simulator, and returns no useful value.

REFERENCES

See also the following general pvar information operators:

allocated–pvar–p	**describe–pvar**	**pvar–exponent–length**
pvar–length	**pvar–mantissa–length**	
pvar–name	**pvarp**	**pvar–plist**
pvar–type	**pvar–vp–set**	

pvar–mantissa–length [*Function*]

Returns bit length of the mantissa of the supplied floating-point or complex pvars.

SYNTAX

pvar–mantissa–length *pvar*

ARGUMENTS

pvar	Floating-point or complex pvar. Pvar for which mantissa bit length is determined.

RETURNED VALUE

exponent–length	Integer. Length in bits of mantissa field of supplied pvar.

SIDE EFFECTS

None.

DESCRIPTION

This function returns the bit-length of the mantissa of *pvar*.

Note: This function has no meaning in the *Lisp simulator, and returns no useful value.

REFERENCES

See also the following general pvar information operators:

allocated–pvar–p	**describe–pvar**	**pvar–exponent–length**
pvar–length	**pvar–location**	
pvar–name	**pvarp**	**pvar–plist**
pvar–type	**pvar–vp–set**	

---

pvar–name *[Function]*

Returns the symbolic name of the supplied pvar.

SYNTAX

pvar–name *pvar*

ARGUMENTS

pvar	Pvar expression. Pvar for which symbolic name is returned.

RETURNED VALUE

name–symbol	Symbol. Symbol recorded as the name of *pvar*.

SIDE EFFECTS

None.

DESCRIPTION

This function returns the symbolic name of *pvar*.

The argument *pvar* may be any pvar, but temporary pvars return **nil**.

REFERENCES

See also the following general pvar information operators:

allocated–pvar–p	**describe–pvar**	**pvar–exponent–length**
pvar–length	**pvar–location**	**pvar–mantissa–length**
pvarp	**pvar–plist**	
pvar–type	**pvar–vp–set**	

pvarp

[Function]

Tests whether the supplied object is a pvar.

SYNTAX

pvarp *object*

ARGUMENTS

object	Common Lisp or *Lisp data object to be tested.

RETURNED VALUE

pvarp	Boolean. The value **t** if *object* is a pvar, and **nil** otherwise.

SIDE EFFECTS

None.

DESCRIPTION

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

REFERENCES

See also the following general pvar information operators:

allocated-pvar-p	**describe-pvar**	**pvar-exponent-length**
pvar-length	**pvar-location**	**pvar-mantissa-length**
pvar-name	**pvar-plist**	**pvar-type**
pvar-vp-set		

pvar–plist [*Function*]

Returns the property list of the supplied pvar.

SYNTAX

pvar–plist *pvar*

ARGUMENTS

pvar	Pvar expression. Pvar for which property list is returned.

RETURNED VALUE

property–list	List. Property list of *pvar*.

SIDE EFFECTS

None.

DESCRIPTION

This function returns the property list of *pvar*. The argument *pvar* may be any pvar.

The "property list" of a pvar is not currently used by *Lisp. It exists so that users may write their own functions to store and access the property lists of pvars. The expression **(setf (pvar–plist** *pvar* **))** may be used to modify the property list slot of a pvar.

REFERENCES

See also the following general pvar information operators:

allocated–pvar–p	**describe–pvar**	**pvar–exponent–length**
pvar–length	**pvar–location**	**pvar–mantissa–length**
pvar–name	**pvarp**	**pvar–type**
pvar–vp–set		

pvar–to–array [*Defun]

Copies values from a pvar to a front-end vector in send-address order.

SYNTAX

pvar–to–array *source–pvar* **&optional** *dest–array*
&key **:array–offset :start end**
:cube–address–start :cube–address–end

ARGUMENTS

source–pvar Pvar. Pvar from which values are copied.

dest–array Front-end vector. Array into which values are stored. If this argument is **nil**, the default, a front-end array of the appropriate size is created and returned.

:array–offset Integer. Offset into *dest–array* at which first value is stored. Default is 0.

:start Send address. Processor at which copying will start. Default is 0.

:end Send address. Processor at which copying will end. Default is ***number–of–processors–limit***.

:cube–address–start :cube–address–end
Obsolete aliases for the **:start** and **:end** keywords, retained for software compatibility only.

RETURNED VALUE

dest–array The destination array, into which values have been copied.

SIDE EFFECTS

The contents of *source–pvar* from **:start** to **:end** are copied into *dest–array* beginning at **:array–offset**.

DESCRIPTION

This function moves data from *source–pvar* into *dest–array* in send-address order.

If provided, *dest–array* must be one-dimensional. If a *dest–array* is not provided, an array is created of size **:end** minus **:start**.

The data from *source–pvar* in processors **:start** through 1 – **:end** are written into the *dest–array* elements starting with element **:array–offset**. The result returned by **pvar–to–array** is *dest–array*.

EXAMPLES

A sample **pvar–to–array** call is the expression

```
(pvar-to-array (self-address!!) nil
   :start 3
   :end 10)
```

which returns the array

```
#(3 4 5 6 7 8 9)
```

A call to **pvar–to–array** that uses the **:array–offset** keyword is

```
(pvar-to-array (self-address!!) nil
   :array-offset 2
   :start 3
   :end 10)
```

which returns the array

```
#(NIL NIL 3 4 5 6 7 8 9)
```

NOTES

Usage Note:

It is an error to supply a value for both **:cube–address–start** and **:start** in the same function call. Likewise, it is an error to provide **:cube–address–end** and **:end** arguments in the same function call.

Performance Note:

The **pvar-to-array** function performs most efficiently when used on non-aggregate pvars of declared type and when the front-end array is of corresponding type to that of the pvar.

For instance, transferring data from a pvar of type **single-float** into an array whose element type is **single-float** is very efficient. Transferring a general pvar into an array whose element type is **t** will not be as efficient.

Transferring aggregate pvars (structures and arrays) using a single call to one of the functions **array-to-pvar**, **pvar-to-array**, **pvar-to-array-grid**, or **array-to-pvar-grid** is very slow. See the performance note under the definition of **array-to-pvar** for a discussion of how to transfer aggregate data efficiently between the front end and the CM.

Syntax Note:

Remember that when no *dest-array* argument is specified to the **pvar-to-array** and **pvar-to-array-grid** functions, a **nil** must be provided instead if keyword arguments are to be used.

REFERENCES

See also these related array transfer operations:

array-to-pvar **array-to-pvar-grid**
pvar-to-array-grid

See also the *Lisp operation **pref**, which is used to transfer single values from the CM to the front end.

The *Lisp operation ***setf**, in combination with **pref**, is used to transfer a single value from the front end to the CM.

pvar–to–array–grid [*Defun]

Copies values from a pvar to a front-end array in grid address order.

SYNTAX

pvar–to–array–grid *source–pvar* **&optional** *dest–array*
&key **:array–offset**
:grid–start
:grid–end

ARGUMENTS

source–pvar Pvar. Pvar from which values are copied.

dest–array Front-end array into which values are stored. Must have a rank equal to ***number–of–dimensions***. If this argument is **nil**, the default, a front-end array of the appropriate size is created and returned.

:array–offset Integer list. Set of offsets into *source–array* indicating location at which first value is stored. Default is **(make–list *number–of–dimensions* :initial–element 0)**.

:grid–start Integer list, specifying *inclusive* grid address of processor at which copying will start. Defaults to the value of the form **(make–list *number–of–dimensions* :initial–element 0)**.

:grid–end Integer list, specifying *exclusive* grid address of processor at which copying will end. Defaults to the value of the variable ***current–cm–configuration***.

RETURNED VALUE

dest–array The destination array, into which values have been copied.

SIDE EFFECTS

The contents of *source–pvar* from **:grid–start** to **:grid–end** are copied into *dest–array* beginning at **:array–offset**.

DESCRIPTION

This function moves data from *source–pvar* into *dest–array* in grid address order.

If provided, *dest–array* must have the same number of dimensions as the current CM configuration. If *dest–array* is not specified, an array is created with dimensions **:grid–end** minus **:grid–start**, where the subtraction is done element-wise to produce a list suitable for **make–array**. The data from *source–pvar* in the sub-grid defined by **:grid–start** and **:grid–end** as the *inclusive* "upper-left" and *exclusive* "lower-right" corners, respectively, are written into a similar sub-grid of *dest–array* starting with element **:array–offset** as the upper-left corner. The arguments **:array–offset**, **:grid–start**, and **:grid–end** must be lists of length ***number–of–dimensions***. The value returned by **pvar–to–array–grid** is *dest–array*.

EXAMPLES

Assuming a two-dimensional grid has been defined, for which

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

displays the values

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

then when the expression

```
(pvar-to-array-grid (self-address!!) nil
   :grid-start '(1 1)  :grid-end '(4 3))
```

is evaluated, it returns the array

```
#2A((5 6) (9 10) (13 14))
```

and the expression

```
(pvar-to-array-grid (self-address!!) nil
   :array-offset '(1 1)
   :grid-start '(1 1)  :grid-end '(4 3))
```

when evaluated, returns the array

```
#2A((NIL NIL NIL) (NIL 5 6) (NIL 9 10) (NIL 13 14))
```

The following example shows the use of **pvar-to-array-grid** to extract a subgrid from a pvar and store it into a predefined front-end array:

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

(defparameter an-array
              (make-array '(10 10)
                          :element-type 'single-float
                          :initial-element 0.0))

(*proclaim '(type single-float-pvar data-pvar))
(*defvar data-pvar (float!! (self-address!!)))

(ppp data-pvar :mode :grid :end '(5 5) :format "~5F ")
     DIMENSION 0 (X)   ----->
   0.0   1.0   2.0   3.0   4.0
   8.0   9.0  10.0  11.0  12.0
  16.0  17.0  18.0  19.0  20.0
  24.0  25.0  26.0  27.0  28.0
 128.0 129.0 130.0 131.0 132.0
```

The following call to **pvar-to-array-grid** transfers the 4 x 4 subgrid of **data-pvar** whose corners are

```
(1 1) (4 1)
(1 4) (4 4)
```

to the 4 x 4 subarray of **an-array** whose corners are

```
(2 3) (6 3)
(2 7) (6 7)

(pvar-to-array-grid data-pvar an-array
                    :array-offset '(2 3)
                    :grid-start '(1 1)
                    :grid-end '(5 5))
(aref an-array 2 3) => 9.0
```

NOTES

Performance Note:

The **pvar-to-array-grid** function performs most efficiently when used on non-aggregate pvars of declared type and when the front-end array is of corresponding type to that of the pvar.

For instance, transferring data from a pvar of type **single–float** into an array whose element type is **single–float** is very efficient. Transferring a general pvar into an array whose element type is **t** will not be as efficient.

Transferring aggregate pvars (structures and arrays) using a single call to one of the functions **array–to–pvar**, **pvar–to–array**, **pvar–to–array–grid**, or **array–to–pvar–grid** is very slow. See the performance note under the definition of **array–to–pvar** for a discussion of how to transfer aggregate data efficiently between the front end and the CM.

Syntax Note:

Remember that when no *dest–array* argument is specified to the **pvar–to–array** and **pvar–to–array–grid** functions, a **nil** must be provided instead if keyword arguments are to be used.

REFERENCES

See also these related array transfer operations:

array–to–pvar **array–to–pvar–grid**
pvar–to–array

See also the *Lisp operation **pref**, which is used to transfer single values from the CM to the front end.

The *Lisp operation ***setf**, in combination with **pref**, is used to transfer a single value from the front end to the CM.

pvar–type *[Function]*

Returns the data type of the supplied pvar.

SYNTAX

pvar–type *pvar*

ARGUMENTS

pvar Pvar expression. Pvar for which data type is determined.

RETURNED VALUE

data–type Symbol representing *Lisp data type for *pvar*.

SIDE EFFECTS

None.

DESCRIPTION

This function returns the data type of *pvar*. The argument *pvar* may be any pvar.

Note: This function always returns the value **t** in the *Lisp simulator.

REFERENCES

See also the following general pvar information operators:

allocated–pvar–p **describe–pvar** **pvar–exponent–length**
pvar–length **pvar–location** **pvar–mantissa–length**
pvar–name **pvarp** **pvar–plist**
pvar–vp–set

---

pvar–vp–set [*Function*]

Returns the VP set to which the supplied pvar belongs.

SYNTAX

pvar–vp–set *pvar*

ARGUMENTS

pvar	Pvar expression. Pvar for which VP set is returned.

RETURNED VALUE

vp–set	*Lisp VP set object. VP set to which *pvar* belongs.

SIDE EFFECTS

None.

DESCRIPTION

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

The argument *pvar* may be any pvar.

REFERENCES

See also the following general pvar information operators:

allocated–pvar–p	**describe–pvar**	**pvar–exponent–length**
pvar–length	**pvar–location**	**pvar–mantissa–length**
pvar–name	**pvarp**	**pvar–plist**
pvar–type		

---

random!! [*Function*]

Returns a pvar with a random value in each processor.

SYNTAX

random!! *limit–pvar*

ARGUMENTS

limit–pvar	Non-complex numeric pvar. Upper exclusive bound on random number selected. Must contain positive values.

RETURNED VALUE

random–pvar	Temporary numeric pvar, of same type as *limit–pvar*. In each active processor, contains a random value between 0 inclusive and the value of *limit–pvar* exclusive.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function is the parallel equivalent of Common Lisp's **random** function, and returns a pvar containing a random value in each processor.

EXAMPLES

For example, when the expression

```
(ppp (random!! (!! 10)) :end 10)
```

is evaluated, the first ten values of the random-valued pvar returned by **random!!** are displayed, for example

```
8 9 1 3 4 0 2 7 6 5
```

NOTES

This operation is faster when provided constant pvar arguments, as in the example above, than when applied to non-constant pvar arguments, as in

```
(*set random-data (random!! data-pvar))
```

rank!! [*Function*]

Performs a parallel comparison, numerically ranking the values of the supplied numeric pvar.

SYNTAX

rank!! *numeric–pvar predicate* **&key :dimension :segment–pvar**

ARGUMENTS

numeric–pvar	Non-complex numeric pvar. Pvar containing values to be compared.
predicate	Two-argument pvar predicate. Determines type of ranking. Currently limited by implementation to the function **<=!!**.
:dimension	Integer or **nil**. Specifies dimension along which to perform ranking. The default, **nil**, specifies a send-address order ranking. If not **nil**, this argument must be an integer between 0 inclusive and ***number–of–dimensions*** exclusive.
:segment–pvar	Segment pvar or **nil**. Specifies segments in which to perform independent rankings. The default, **nil**, specifies an unsegmented ranking.

RETURNED VALUE

rank–pvar	Temporary integer pvar. In each active processor, contains the numeric rank of the corresponding value of *numeric–pvar* among all of the active values of *numeric–pvar*, under the relation specified by *predicate*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The **rank!!** function returns a pvar containing values from 0 through one less than the number of active processors. The order of the values in the returned *rank–pvar* indicates the ranking of the values in the supplied *numeric–pvar.*

The ranking is performed so that for any two active processors *p1* and *p2*, if the value of *rank–pvar* in *p1* is less than the value of *rank–pvar* in *p2*, then the value of *numeric–pvar* in processor *p1* satisfies the supplied *predicate* with respect to the value of *numeric–pvar* in processor *p2*. (The current implementation limits *predicate* to the operator **<=!!**.)

The keywords, **:dimension** and **:segment–pvar** permit rankings to be taken along specific grid dimensions and within segments.

The **:dimension** keyword specifies whether the ranking is done by send address order or along a specific dimension. If a dimension is specified, ranking is performed only along that dimension. The default value, **nil**, specifies a send-address order ranking.

For example, assuming a two-dimensional grid, a **:dimension** argument of 0 causes ranking to occur independently in each "row" of processors along dimension 0. A **:dimension** argument of 1 causes ranking to occur independently in each "column" of processors along dimension 1 (see Figure 2).

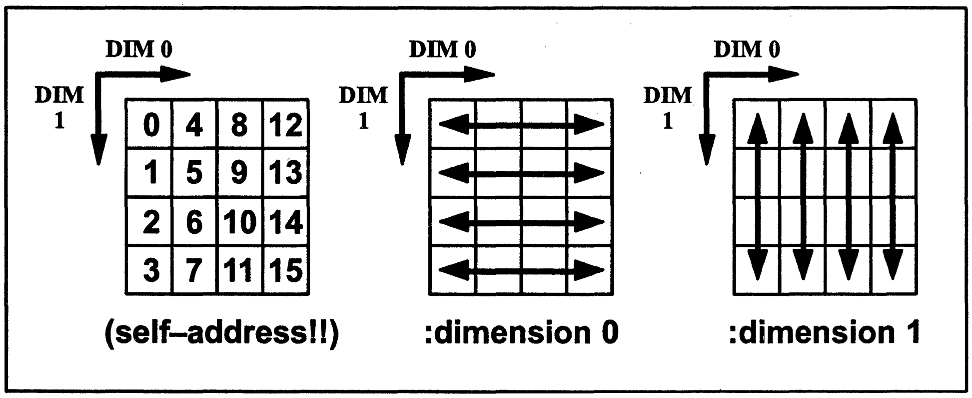

Figure 2. Effect of different **:dimension** arguments, assuming a two-dimensional grid.

The **:segment–pvar** argument specifies whether the ranking is performed separately within segments. The default is **nil**; **rank!!** is by default unsegmented. If provided, the **:segment–pvar** value must be a segment pvar. A segment pvar contains boolean values, with a non-**nil** value in the first processor of each segment and **nil** in all other processors. If a segment pvar is specified, then the ranking is done independently within each segment.

If both a **:dimension** and a **:segment pvar** argument are specified, then the ranking is done independently for each "row" along the specified dimension and independently within segments for each row.

EXAMPLES

A simple call to **rank!!** is

```
(rank!! numeric-pvar '<=!!)
```

If the first 12 elements of **numeric-pvar** are

```
0 20 4 16 8 12 10 14 6 18 2 22
```

then the first 12 values of the returned *rank-pvar* are

```
0 10 2 8 4 6 5 7 3 9 1 11
```

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

```
(rank!! numeric-pvar '<=!!
        :segment-pvar (evenp!! (self-address!!)))
```

If the first 12 elements of **numeric-pvar** are

```
0  2  4  2  1  7  5  3  4  7  8  2
```

then the first 12 values of the returned *rank-pvar* are

```
0  1  1  0  0  1  1  0  0  1  1  0
```

An example of **rank!!** with a **:dimension** argument is

```
(rank!! (self-address!!) '<=!! :dimension 1)
```

Assuming a two-dimensional VP set geometry, if the expression

```
(*defvar random-values (random!! (!! 32)))
(ppp random-values :mode :grid :end '(4 4))
```

displays the values

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

then the expression

```
(ppp     (rank!! random-values '<=!! :dimension 1)
         :mode :grid :end '(4 4))
```

will display the values

```
0  3  0  3
1  2  2  1
2  1  1  2
3  0  3  0
```

The function **sort!!** might be implemented using a combination of **rank!!** and ***pset**, as follows:

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

(*defvar random-values (random!! (!! 32)))
(ppp random-values)
22 17 5 31 0 4 12 4

(defun my-sort!! (unsorted-pvar)
   (*let (sorted-pvar)
      (*pset :no-collisions unsorted-pvar sorted-pvar
                            (rank!! sorted-pvar)
      sorted-pvar)))
```

NOTES

The ranking performed by **rank!!** is not guaranteed to be stable. If *numeric–pvar* contains the same value in two or more active processors, the ordering returned for these values in *rank–pvar* is arbitrary and indeterminate.

Compiler Note:

The *Lisp compiler does not compile **rank!!** if a **:segment–pvar** argument is supplied.

REFERENCES

See also the related functions

enumerate!!	**self!!**	
self-address!!	**self-address-grid!!**	**sort!!**

realpart!! *[Function]*

Extracts the real component from a complex pvar.

SYNTAX

realpart!! *numeric–pvar*

ARGUMENTS

numeric–pvar Numeric pvar. Pvar from which real part is extracted.

RETURNED VALUE

realpart–pvar Temporary numeric pvar. In each active processor, contains the real part of the corresponding value of *numeric–pvar.*

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function returns a temporary pvar containing in each processor the real component of the complex value in *numeric–pvar.* Note that *numeric–pvar* need not be explicitly a complex-valued pvar. Non-complex values are automatically coerced into complex values with a zero imaginary component. Note that you can apply ***setf** to an **imagpart!!** call to modify the imaginary component of a complex numeric pvar.

REFERENCES

See also these related complex pvar operators:

abs!!	**cis!!**	**complex!!**
conjugate!!	**imagpart!!**	**phase!!**

reduce!! [Function]

Combines elements of a sequence pvar in parallel using a binary pvar function.

SYNTAX

reduce!! *function sequence–pvar* **&key :from–end :start :end :initial–value**

ARGUMENTS

function	Two-argument pvar function. Used to combine elements of *sequence–pvar* in parallel.
sequence–pvar	Sequence pvar. Pvar containing sequences to be reduced.
:from–end	Scalar boolean. Whether to begin search from end of sequence. Defaults to **nil**.
:start	Integer pvar. Index, zero-based, of sequence element at which reduction operation starts. If not specified, search begins with first element.
:end	Integer pvar. Index, zero-based, of sequence element at which reduction operation ends. If not specified, search continues to end of sequence.
:initial–value	Pvar, of same type as elements of *sequence–pvar*. If supplied, is included in reduction operation as first value supplied to *function*.

RETURNED VALUE

reduce–pvar	Temporary pvar, of same type as elements of *sequence–pvar*. In each active processor, contains result of reducing the corresponding sequence of *sequence–pvar* by the supplied *function*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The function **reduce!!** is similar to the Common Lisp function **reduce**. It 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.

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** define a subsequence of *sequence–pvar*.

The keyword **:initial–value** takes a pvar of the same type as the elements of *sequence–pvar* and provides an initial value for the reduction calculation. If an **:initial–value** value is supplied, it is logically placed at the beginning of *sequence–pvar* and included in the reduction. If **:from–end** is **t**, the value of **:initial–value** is logically placed at the end of *sequence–pvar*.

EXAMPLES

The expression

```
(reduce!! #'+!! number-sequence-pvar)
```

adds up the elements of **number–sequence–pvar** in each processor.

NOTES

Language Note:

Although the function **reduce!!** is in many way similar to the Common Lisp function **reduce**, it is not exactly identical, for while **reduce** can return any Common Lisp value, **reduce!!** can only return a pvar of the same type as the elements of *sequence–pvar.*

Compiler Note:

Because of the utility of the **reduce!!** function for vector pvar operations, the *Lisp compiler will compile this function, but only under certain conditions. Specifically, for **reduce!!** to compile, the *function* argument must be a compilable function, and none of the keyword arguments may be used.

REFERENCES

See also these related *Lisp sequence operators:

copy–seq!!	***fill**	**length!!**
***nreverse**	**reverse!!**	**subseq!!**

See also the generalized array mapping functions **amap!!** and ***map**.

reduce-and-spread!! *[Function]*

Performs a **scan!!** reduction along the specified dimension of the currently defined grid, and then a backwards **copy!!** scan to spread the result values to all processors along the scanned dimension.

SYNTAX

reduce-and-spread!! *pvar function dimension*

ARGUMENTS

pvar	Pvar expression. Pvar containing values to be reduced.
function	Two-argument pvar function. Determines type of reduction. May be any of **+!!**, **and!!**, **or!!**, **logand!!**, **logior!!**, **logxor!!**, **max!!**, **min!!**, and **copy!!**.
dimension	Integer or **nil**. Index, zero-based, of dimension of currently defined grid along which reduction is performed, and the result values are copied. A value of **nil** indicates that a send-address reduction and spread should be performed.

RETURNED VALUE

scan–pvar	Temporary pvar. A copy of *pvar* to which the reduction operation specified by *function* has been applied, with the result spread to every processor along the dimension specified by *dimension.*

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

Conceptually, this function first performs a

```
(scan!! pvar function :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 *dimension* argument determines the grid dimension along which the operation is performed. It must be either a non-negative integer scalar within the range of dimensions of the VP set to which *pvar* belongs, or **nil**. If *dimension* is **nil**, send-address order scanning is done.

For example, assuming a two-dimensional grid, a *dimension* argument of 0 causes ranking to occur independently in each "row" of processors along dimension 0. A *dimension* argument of 1 causes ranking to occur independently in each "column" of processors along dimension 1 (see Figure 3). Because the grid has only two dimensions, the only valid arguments for *dimension* are 0, 1, and **nil**.

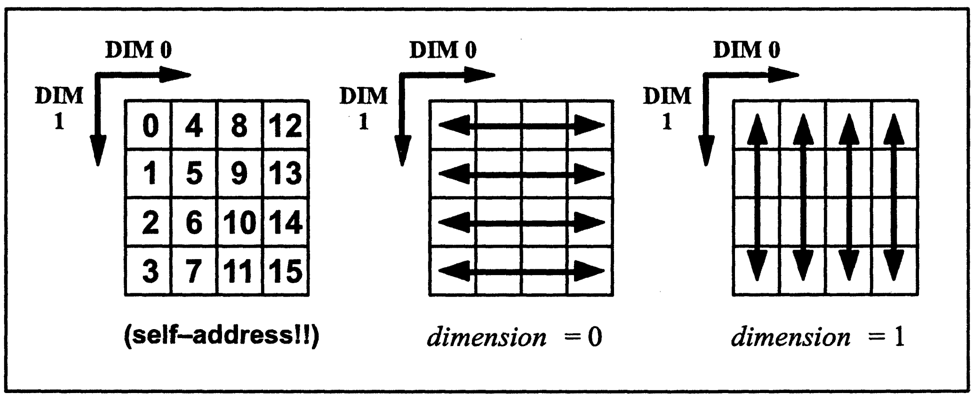

Figure 3. Effect of different *dimension* arguments, assuming a two-dimensional grid.

EXAMPLES

This example shows how **reduce–and–spread** may be used, assuming a two-dimensional grid configuration for simplicity. Note that the reduction and spread operation is performed along dimension 1, that is, down the "columns" of the grid.

```
(*cold-boot :initial-dimensions '(4 4))

(ppp (self-address!!) :mode :grid :format "~2D ")

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

(ppp  (reduce-and-spread!! (self-address!!) '+!! 1)
      :mode :grid :format "~2D ")

 6 22 38 54
 6 22 38 54
 6 22 38 54
 6 22 38 54
```

NOTES

Performance Note:

This function is provided because it may be significantly faster to use it than to do a **scan!!** followed by a reverse copy scan.

REFERENCES

See also these related operations:

scan!! **segment–set–scan!!** **spread!!**

rem!! [Function]

Calculates in parallel the remainder of a division on the supplied pvars.

SYNTAX

rem!! *numeric–pvar divisor–pvar*

ARGUMENTS

numeric–pvar	Non-complex numeric pvar. Pvar for which remainder is calculated.
divisor–pvar	Integer pvar. Pvar by which *numeric–pvar* is divided.

RETURNED VALUE

remainder–pvar	Temporary numeric pvar, of same type as *numeric–pvar*. In each active processor, contains the remainder from dividing the value of *numeric–pvar* by the value of *divisor–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function is the parallel equivalent of the Common Lisp function **rem**. It is an error if *divisor–pvar* contains zero in any processor.

---

reverse!! [Function]

Returns a copy of the supplied sequence pvar in which each sequence has been reversed.

SYNTAX

reverse!! *sequence–pvar*

ARGUMENTS

sequence–pvar Sequence pvar. Pvar containing sequences to be reversed.

RETURNED VALUE

reverse–pvar Temporary sequence pvar. In each active processor, contains a reversed copy of the corresponding sequence of *sequence–pvar.*

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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))
```

NOTES

Compiler Note:

The *Lisp compiler does not compile this operation.

REFERENCES

See also these related *Lisp sequence operators:

copy–seq!!	***fill**	**length!!**
***nreverse**	**reduce!!**	**subseq!!**

See also the generalized array mapping functions **amap!!** and ***map**.

*room [Function]

Prints and returns information about CM memory use.

SYNTAX

***room &key :how :print-statistics :stream**

ARGUMENTS

:how	One of **:by-vp-set**, **:by-pvar**, or **:totals**. Specifies how usage information is to be displayed. Default is **:by-vp-set**.
:print-statistics	Scalar boolean. Whether to print results as well as returning values. Defaults to **t**.
:stream	Stream object or **t**. Stream to which results are printed. Defaults to **t**, sending output to ***standard-output*** stream.

RETURNED VALUES

stack-bytes	Integer. Number of bits of CM memory in use by temporary pvars on the *Lisp stack.
temp-bytes	Integer. Number of bits of CM memory in use by permanent pvars on the *Lisp heap that were created by **allocate!!**.
defvar-bytes	Integer. Number of bits of CM memory in use by permanent pvars on the *Lisp heap that were created by ***defvar**.
overhead-bytes	Integer. Number of bits of CM memory in use as overhead.

SIDE EFFECTS

None.

DESCRIPTION

Collects and prints information about CM memory usage.

The ***room** function returns four values. Each return value indicates the total amount of CM memory in use for a particular purpose at the time of the call.

> The first return value reports the total number of bits of CM memory allocated on the *Lisp stack.
>
> The second return value reports the total number of bits of CM memory on the heap allocated to pvars created with **allocate!!**.
>
> The third return value reports the total number of bits of CM memory on the heap allocated to pvars created with ***defvar**.
>
> The fourth return value reports the total number of bits of CM memory in use as overhead, including overhead for the *Lisp VP mechanism and overhead for Paris.

The **:how** keyword argument must be either **:by–vp–set** (the default), **:by–pvar**, or **:totals**. If the value of **:how** is **:by–vp–set**, then the four statistics are collected and printed for each existing *Lisp VP set. If the value of **:how** is **:by–pvar**, then statistics are given for each pvar as well as for each VP set. If the value of **:how** is **:totals**, then only summary information is printed. The **:how** keyword argument specifies only how memory information is printed; it has no impact on the values returned by ***room**.

The **:print–statistics** keyword defaults to **t**. If it is set to **nil**, the results are returned but not printed and the **:how** keyword is ignored.

The **:stream** keyword defaults to **t**, indicating that output goes to the standard output device. An alternate stream may be specified.

rot!! [*Function*]

Performs a parallel bit rotation on the supplied integer pvar.

SYNTAX

rot!! *integer–pvar n–pvar word–size*

ARGUMENTS

integer–pvar	Integer pvar. Pvar containing values to be rotated.
n–pvar	Integer pvar. Number of bits to rotate *integer–pvar*. Positive value rotates towards high-order bits, negative towards low-order bits.
word–size	Integer pvar. Number of low-order bits of *integer–pvar* that are rotated.

RETURNED VALUE

rot–pvar	Temporary integer pvar. In each active processor, contains a copy of the low-order *word–size* bits of *integer–pvar* rotated the number of bits specified by the value of *n–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function returns *integer–pvar* rotated left *n–pvar* bits, or rotated right if *n–pvar* is negative. The rotation considers each value of *integer–pvar* to be an integer of length *word–size* bits.

NOTES

This function is especially fast when *n–pvar* and *word–size* are both constant pvars.

---

round!! *[Function]*

Performs a parallel round operation on the supplied pvar(s).

SYNTAX

round!! *numeric–pvar* **&optional** *divisor–numeric–pvar*

ARGUMENTS

numeric–pvar Non-complex numeric pvar. Value to be rounded.

divisor–numeric–pvar
Non-complex numeric pvar. If supplied, *numeric–pvar* is divided by *divisor–numeric–pvar* before rounding.

RETURNED VALUE

round–pvar Temporary integer pvar. In each active processor, contains the rounded value of *numeric–pvar*, divided by *divisor–numeric–pvar* if supplied.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This is the parallel equivalent of the Common Lisp function **round**, except that only one value (the rounded quotient) is computed and returned. The **round!!** function rounds numbers to the nearest integer. If a number is exactly halfway between two integers, it is rounded towards the even integer.

REFERENCES

See also these related rounding operations:

ceiling!! **floor!!** **truncate!!**

See also these related floating-point rounding operations:

fceiling!! **ffloor!!** **fround!!** **ftruncate!!**

row–major–aref!! *[Function]*

References the supplied multidimensional array pvar as a vector pvar with elements in row-major order.

SYNTAX

row–major–aref!! *array–pvar row–major–index–pvar*

ARGUMENTS

array–pvar Array pvar. Pvar to be referenced.

row–major–index–pvar
Integer pvar. Index of element in *array–pvar* to retrieve.

RETURNED VALUE

row–major–aref–pvar
Temporary pvar, of same type as elements of *array–pvar*. In each active processor, contains the element of *array–pvar* at the location referenced by *row–major–index–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

References the specified array pvar as if it were a vector pvar, with elements taken in row-major order. The result is returned as a pvar.

The *array–pvar* argument may be any array pvar. If this is a vector pvar (a one-dimensional array pvar), then this function is equivalent to **aref!!**.

The *row–major–index–pvar* must contain integers in the range [0...*N*], where *N* is one less than the total number of elements in *array–pvar*. In each processor, this value specifies the row-major index of a single element in the component array.

EXAMPLES

Consider the following:

```
(*defvar my-array (!! #2A((5 8) (3 0))))
(pref (row-major-aref!! my-array (!! 2)) 19) => 3
```

In each processor is stored the array:

```
5 8
3 0
```

The element with row-major index 2 is referenced using **row–major–aref!!**. This results in a pvar whose value is 3 everywhere. The **pref** function then references this value in the 19th processor, yielding 3.

It is legal to compose ***setf** with **row–major–aref!!**. For example,

```
(*setf (row-major-aref!! my-array (!! 2)) (!! 25))
```

stores the value 25 in the third element of the component array in each processor.

```
(pref (row-major-aref!! my-array (!! 2)) 19) => 25
```

NOTES

Usage Note:

The **row–major–aref!!** function can be used to implement subroutines that perform operations on arrays of any dimensionality.

REFERENCES

See also the related array-referencing operations:

aref!! **row–major–sideways–aref!!** **sideways–aref!!**

The following operations convert arrays to and from sideways orientation:

***processorwise** ***sideways–array** ***slicewise**

See also the ***map** and **amap!!** functions for another way to iterate in row-major order over the elements of array pvars of any dimensionality.

row–major–sideways–aref!! [*Function*]

References the supplied multidimensional sideways (slicewise) array pvar as a vector pvar with elements in row-major order.

SYNTAX

row–major–sideways–aref!! *array–pvar row–major–index–pvar*

ARGUMENTS

array–pvar Sideways array pvar. Pvar to be referenced.

row–major–index–pvar
Integer pvar. Index of element in *array–pvar* to retrieve.

RETURNED VALUE

row–major–aref–pvar
Temporary pvar, of same type as elements of *array–pvar*. In each active processor, contains the element of *array–pvar* at the location referenced by *row–major–index–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

References the specified sideways (slicewise) array pvar as if it were a vector pvar, with indices taken in row-major order. The result is returned as a pvar.

The *row–major–index–pvar* must contain integers in the range [0..*N*], where *N* is one less than the number of elements in *array–pvar*. In each processor, this value specifies the row-major index of a single element in the component array.

EXAMPLES

Consider the following:

```
(*proclaim '(type (array-pvar (unsigned-byte 8) '(2 2))
             my-sideways-array))
(*defvar my-sideways-array (!! #2A((5 8) (3 0))))
```

In each processor is stored the array: 5 8
3 0

The array is turned sideways, and is verified to be sideways.

```
(*slicewise my-sideways-array)
(sideways-array-p my-sideways-array)  =>  T
```

In the following example, a different index into **my–sideways–array** is calculated in each processor, and then the array elements corresponding to those indices are accessed using **row–major–sideways–aref!!**.

```
(ppp (row-major-sideways-aref!! my-sideways-array
         (mod!! (self-address!!) (!! 4)))
     :end 14)

5 8 3 0 5 8 3 0 5 8 3 0 5 8
```

It is legal to compose ***setf** with **row–major–sideways–aref!!**. For example,

```
(*setf (row-major-sideways-aref!! my-sideways-array
            (!! 2))
       (!! 25))
```

stores the value 25 in the third element of the component array in each processor.

```
(ppp  (row-major-sideways-aref!! my-sideways-array
          (mod!! (self-address!!) (!! 4)))
      :end 14)

5 8 25 0 5 8 25 0 5 8 25 0 5 8
```

REFERENCES

See also the related array-referencing operations:

aref!! **row–major–aref!!**
sideways–aref!!

The following operations convert arrays to and from sideways orientation:

***processorwise** ***sideways–array** ***slicewise**

sbit!! *[Function]*

Selects in parallel a bit at a given location in a simple bit array pvar.

SYNTAX

sbit!! *bit–array–pvar* **&rest** *pvar–indices*

ARGUMENTS

bit–array–pvar	Simple bit array pvar. Array from which bit is selected.
pvar–indices	Integers. Must be valid subscripts for *bit–array–pvar.* Specifies location of bit to return.

RETURNED VALUE

bit–pvar	Temporary bit pvar. In each processor, contains the bit retrieved from the corresponding array of *bit–array–pvar.*

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function returns a temporary pvar whose value in each processor is the element of the bit-array in *bit–array–pvar* referenced by *pvar–indices.* This function is similar to **bit!!**, but *bit–array–pvar* is expected to be a simple array, i.e., a non-displaced, static array that has no fill pointer.

Note: There is no significant efficiency advantage to using this function in place of **aref!!**; the two are equivalent. Furthermore, you should use **aref!!** instead because **sbit!!** will not exist in future versions of *Lisp.

---

scale–float!! [*Function*]

Multiplies the supplied floating-point pvar by the specified power of two.

SYNTAX

scale–float!! *float–pvar power–of–two–pvar*

ARGUMENTS

float–pvar	Floating-point pvar. Pvar to be scaled.
power–of–two–pvar	Integer pvar. Power of two by which *float–pvar* is scaled.

RETURNED VALUE

scale–float–pvar	Temporary floating-point pvar. In each active processor, contains the corresponding value of *float–pvar* multiplied by two to the power specified by *power–of–two–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function takes a floating-point pvar and an integer pvar; it returns, in each processor, that processor's *float–pvar* component multiplied by two to that processor's *power–of–two–pvar* component power.

EXAMPLES

```
(scale-float!! (!! 3.5) (!! -1)) <=> (!! 1.75)
(scale-float!! (!! 1.0) (!! 2))  <=> (!! 4.0)
```

scan!! *[Function]*

Performs a cumulative reduction operation on the supplied pvar, either by send address or along a specified dimension of the currently defined grid.

SYNTAX

scan!! *pvar function* **&key :direction :segment–pvar :segment–mode**
:include–self :dimension :identity

ARGUMENTS

pvar	Pvar expression. Pvar containing values to be scanned.
function	Two-argument pvar function. Determines type of scan. May be any of **+!!**, ***!!**, **and!!**, **or!!**, **logand!!**, **logior!!**, **logxor!!**, **max!!**, **min!!**, and **copy!!**, or a user-defined function, in which case a value must be supplied for the **:identity** argument.
:direction	Either **:forward** or **:backward**. Determines direction of scan through send addresses or across grid. Default is **:forward**.
:segment–pvar	Boolean pvar containing the value **t** in each processor that starts a segment, and the value **nil** elsewhere. Determines segments within which scanning takes place. If not supplied, an unsegmented scan is performed.
:segment–mode	Either **:start**, **:segment**, or **nil**. Controls whether the **:segment–pvar** argument is evaluated in all processors or only active ones.
:include–self	Boolean. Determines whether to include the value contained in each processor in the scan calculation for that processor. Default is **t**.
:dimension	Integer. Index, zero-based, of dimension of currently defined grid along which scanning is performed. If not supplied, a send-address order scan is performed.
:identity	Scalar. Identity element for *function*. Must be supplied if *function* is not a specialized scanning function. Ignored otherwise.

RETURNED VALUE

scan–pvar Temporary pvar. A copy of *pvar* to which the scanning operation specified by *function* has been applied.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The **scan!!** function performs a cumulative reduction operation on the supplied pvar, either by send address or along one dimension of the currently defined grid.

"Reducing" in this context refers to the Common Lisp function **reduce**, which accepts two arguments, *function* and *sequence*. The **reduce** function applies *function*, which must be a binary associative function, to all the elements of the *sequence*. For example, if **+** were the *function* all the elements in *sequence* would be summed. In the case of a **scan!!** function, the sequence becomes the pvar values contained in the ordered set of selected processors.

For each selected processor, the value returned to that processor is the result of reducing the pvar values in all the processors preceding it. Its own *pvar* value is also, by default, included in the reduction.

The *function* argument may be one of the associative binary pvar functions **+!!**, **and!!**, **or!!**, **logand!!**, **logior!!**, **logxor!!**, **max!!**, **min!!**, or **copy!!**, in which case an efficient "specialized scan" is performed. In addition, other associative binary pvar operators may be supplied, including user-defined pvar functions, in which case a less efficient "generalized scan" is performed.

The function ***!!** is a special case; if used to perform a scan on a floating-point pvar, it performs as efficiently as one of the specialized scan operators listed above. If applied to any other numeric arguments, it is treated as a generalized scan operator.

The **:direction** keyword controls the direction of the scan through send addresses or across the grid. The default value for this argument, the keyword **:forward**, causes the scan to be performed in order of ascending send or grid addresses. The keyword **:backword** causes the scan to be performed in descending order.

The **:segment–pvar** argument provides a limited segmented scan functionality, which permits independent scans to be perfomed within mutually exclusive groups of processors, known as "segments." It must be a boolean pvar containing the value **t** in each processor that starts a segment, and **nil** elsewhere. The end of each segment is determined by the starting point of the next segment. More advanced segmented scans, in particular scans with non-contiguous segments, are possible through the function **segmented–set–scan!!**.

If **:segment–pvar** is provided, and **:segment–mode** is given the value **:segment**, then the segment pvar for the **scan!!** operation is interpreted in all processors without respect to the currently selected set. If **:segment–mode** is given the value **:start**, the segment pvar is examined only in those processors that are currently active.

The boolean keyword argument **:include–self** controls whether the scan result calculated in each processor includes the value of *pvar* in that processor. When **:include–self** is **nil**, the result of the **scan!!** operation is undefined in the first active processor of the first segment. Also, when **:include–self** is **nil**, the result of the **scan!!** operation in the first processor of each of the other segments is the cumulative result of the **scan!!** operation over all active processors in the immediately preceding segment.

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, the expression

```
(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*.

If a generalized scan is performed, an **:identity** keyword value must be supplied. If supplied, the value of **:identity** must be the identity value for *function*. That is, if *function* is applied to the pvar (**!!** *identity*) and any legal *pvar* value *P*, the result is *P*. It is an error to specify the **:identity** keyword for specialized scans.

EXAMPLES

If *function* is the function **+!!**, **scan!!** performs a summation over the set of selected processors, ordered by cube address as shown below:

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

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

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

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

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

Following are two further examples using **+!!** with segmented scans. (The "*" indicates a pvar value that is not defined.)

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

The use of the keyword argument **:include-self** with a value of **nil** prevents each processor from including its own value for **(self-address!!)** in the scan. Note that the result of the scan is not defined for processor 0 in the second scan example, and that result of the scan in the first processor of each of the other segments is the cumulative sum of the values in the immediately preceding segment.

The next example, using the **max!!** function, illustrates the double effect achieved when **:include-self** is **nil**. (Again, the "*" indicates a pvar value that is not defined.)

```
pvar                          => 1  10   5   20  3   4  5   6 ...
segment-pvar                  => t  nil nil  t   t  nil nil t ...
(scan!! pvar 'max!!
   :segment-pvar segment-pvar
   :include-self t)           => 1  10  10   20  3   4  5   6 ...
(scan!! pvar 'max!!
   :segment-pvar segment-pvar
   :include-self nil)         => *   1  10   10  20  3  4   5 ...
```

The next example demonstrates the used of **copy!!** with segmented scans:

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

The **scan!!** function can also be used to perform scans on multi-dimensional grids. For example, assuming a two-dimensional grid is defined for which the expression

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

displays the values

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

then the expression

```
(ppp  (scan!! (self-address!!) '+!! :dimension 0)
      :mode :grid :end '(4 4))
```

displays the values

```
0  4 12 24
1  6 15 28
2  8 18 32
3 10 21 36
```

and the expression

```
(ppp (scan!! (self-address!!) '+!! :dimension 1)
    :mode :grid :end '(4 4))
```

displays the values

```
0  4  8 12
1  9 17 25
3 15 27 39
6 22 38 54
```

The following example shows a segmented backwards **copy!!** scan along dimension 1 of the grid with an **:include-self** value of **nil**. If the expression

```
(ppp (self-address!!) :mode :grid :end '(4 5))
```

displays the values

```
0  5  10  15
1  6  11  16
2  7  12  17
3  8  13  18
4  9  14  19
```

then

```
(ppp (scan!! (self-address!!) 'copy!!
             :dimension 1
             :direction :backwards
             :segment-pvar (evenp!! (self-address-grid!! (!! 1)))
             :include-self nil)
   :mode :grid
   :end '(4 4))
```

displays the values

```
2  7  12  17
2  7  12  17
4  9  14  19
4  9  14  19
```

The **:segment-mode** keyword corresponds directly to the *smode* argument of the Paris **cm:scan-with-...** operators. (See the discussion of the *smode* argument on pp. 35–38 of the *Connection Machine Parallel Instruction Set (Paris) Reference Manual.*) This feature allows one to divide the virtual processors into segments via a segment pvar, and then perform scans on those segments without worrying about whether the processors containing the segment bits in the segment pvar are actually in the currently selected set.

The **:segment-mode** argument defaults to **:start** if a **:segment-pvar** argument is provided. This default behavior is consistent with the semantics of **scan!!** in previous releases.

If no **:segment-pvar** argument is provided, **:segment-mode** defaults to **nil**, and has no effect on the **scan!!** operation.

The difference between the **:start** and **:segment** values for the **:segment-mode** argument is illustrated by the following function:

```
(defun difference-between-segment-and-start ()
 (*let ((source (self-address!!)) dest segment)
  (declare (type (signed-pvar *current-send-addresslength*)
           source dest))
  (declare (type boolean-pvar segment))
  (*set segment (evenp!! (self-address!!)))
  (*set dest (!! -1))
  (*when (not!! (=!! (!! 2) (mod!! (self-address!!) (!! 4))))
     (*set dest
        (scan!! source '+!!  :segment-pvar segment
                             :segment-mode :start))
     (ppp dest :end 4)
     (*all (*set dest (!! -1)))
     (*set dest
        (scan!! source '+!!  :segment-pvar segment
                             :segment-mode :segment))
     (ppp dest :end 4))))
```

A sample call to this function looks like:

```
(difference-between-segment-and-start)
0 1 -1 4
0 1 -1 3
```

In the first scan, because processor 2 (counting from 0) is not in the currently selected set, the fact that there is a **t** in that processor in the **segment** pvar is ignored, and the scan segment extends over processors 0, 1, 2 and 3. (Processor 2, being deselected, does not receive a value). Processor 3 receives the sum of the values 0, 1 and 3, i.e., 4.

In the second scan, with **:segment-mode :segment**, even though processor 2 is not enabled, the fact that the **segment** pvar has a **t** value within it is recognized, and the first four processors are broken into two scan segments, 0,1 and 2,3. Processor 3 only receives the sum of the value in processor 3 now (because processor 2 is disabled).

Finally, an example of a "generalized" scan is the following expression. A function that performs 2 *x* 2 parallel matrix multiplication is supplied as the value of *function*, and specifies the identity matrix as the **:identity** argument.

```
(scan!! my-parallel-matrix 'my-matmult2x2!!
    :identity (make-array '(2 2) :initial-contents '((1 0) (0 1))))
```

NOTES

Usage Notes:

Because operations defined by ***defun** are actually macros in disguise (see the entry on ***defun**), ***defun** operations will not work as *function* arguments to **scan!!**. If possible, use **defun** to define these operations instead, or use **defun** to create a function that calls the ***defun** you wish to use.

Performance Notes:

Providing a generalized function to **scan!!** results in significantly slower performance than providing one of the standard, specialized functions.

Scans are performed essentially in constant time. However, at high VP ratios scan performance is improved because of the high number of sends performed between virtual processors located on the same physical chip.

Compiler Note:

Generalized scans do not compile.

REFERENCES

See also these related operations:

create-segment-set!!	**reduce-and-spread!!**
segment-set-scan!!	**spread!!**

segment–set–end–address {–bits}
segment–set–processor–not–in–any–segment
segment–set–start–address {–bits} [*Function*]

Return information about a segment set structure object.

SYNTAX

segment–set–end–address *segment–set–object*
segment–set–end–bits *segment–set–object*
segment–set–processor–not–in–any–segment *segment–set–object*
segment–set–start–address *segment–set–object*
segment–set–start–bits *segment–set–object*

ARGUMENTS

segment–set–object
Segment set structure object (any single value from a segment set pvar created by **create–segment–set!!**).

RETURNED VALUE

Each of the above functions returns a single value, as described below:

end–address Integer. Send address of last processor in segment to which *segment–set–object* belongs.

end–bits Boolean. The value **t** if this segment set object is the last in its segment, and the value **nil** if not.

processor–not–in–any–segment
Boolean. The value **t** if this segment set object is not a member of any segment in its segment set, and the value **nil** if not.

start–address Integer. Send address of first processor in segment to which *segment–set–object* belongs.

start–bits Boolean. The value **t** if this segment set object is the first in its segment, and the value **nil** if not.

SIDE EFFECTS

None.

DESCRIPTION

These are the scalar versions of the corresponding parallel segment set accessor functions. They take a segment set object, as returned by **create–segment–set!!**, and return information about it, as described in the Returned Value section above.

REFERENCES

For information about segment set structure objects, see the dictionary entry for **create–segment–set!!**.

See also these related segment set operators:

segment–set–scan!!
segment–set–end–address!!
segment–set–end–bits!!
segment–set–processor–not–in–any–segment!!
segment–set–start–address!!
segment–set–start–bits!!

segment-set-end-address!! {-bits!!}
segment-set-processor-not-in-any-segment!!
segment-set-start-address!! {-bits!!} [*Function*]

Returns in parallel information about the supplied segment set pvar.

SYNTAX

segment-set-end-address!! *segment-set-pvar*
segment-set-end-bits!! *segment-set-pvar*
segment-set-processor-not-in-any-segment!! *segment-set-pvar*
segment-set-start-address!! *segment-set-pvar*
segment-set-start-bits!! *segment-set-pvar*

ARGUMENTS

segment-set-pvar Segment set pvar, as returned by **create-segment-set!!**.

RETURNED VALUE

Each of these functions returns a single temporary pvar, as described below:

end-address-pvar In each active processor, send address of last processor in segment to which the processor belongs.

end-bits-pvar In each active processor, the value **t** if the processor is the last in its segment, and the value **nil** if not.

processor-not-in-any-segment-pvar
In each active processor, contains the value **t** if processor is not a member of any segment, and the value **nil** otherwise.

start-address-pvar In each active processor, send address of first processor in segment to which the processor belongs.

start-bits-pvar In each active processor, the value **t** if the processor is the first in its segment, and the value **nil** if not.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

These functions take a segment set pvar, as returned by **create-segment-set!!**, and return information about it, as described in the Returned Value section above.

REFERENCES

For information about the components of a segment set structure pvar, see the dictionary entry for **create-segment-set!!**.

See also these related segment set operators:

segment-set-scan
segment-set-end-address!!
segment-set-end-bits!!
segment-set-processor-not-in-any-segment!!
segment-set-start-address!!
segment-set-start-bits!!

segment–set–scan!! [*Function*]

Within the segment sets defined by the supplied segment set pvar, performs a cumulative reduction operation on the supplied pvar, as with the function **scan!!**.

SYNTAX

segment–set–scan!! *pvar scan–operator segment–set–pvar*
    &key **:direction**
        :check–for–processors–not–in–segment–set
        :activate–all–processors–in–segment–set

ARGUMENTS

pvar — Pvar expression. Pvar containing values to be reduced.

function — Two-argument pvar function. Determines type of reduction. May be any of **+!!**, **and!!**, **or!!**, **logand!!**, **logior!!**, **logxor!!**, **max!!**, **min!!**, and **copy!!**.

segment–set–pvar — Segment set pvar, as returned by **create–segment–set!!**. Determines segments within which scanning takes place.

:direction — Either **:forward** or **:backward**. Determines direction of scan through send addresses or across grid. Default is **:forward**.

:check–for–processors–not–in–segment–set
Boolean. Whether to signal an error if *segment–set–pvar* includes processors that are not defined to be in any segment.

:activate–all–processors–in–segment–set
Boolean. Whether to temporarily bind currently selected set so that all processors included in a segment of *segment–set–pvar* are active for duration of scan.

RETURNED VALUE

scan–pvar — Temporary pvar. A copy of *pvar* to which the scanning operation specified by *function* has been applied.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

A **segment-set-scan!!** operation works the same way as the **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.

Unlike **scan!!**, **segment-set-scan!!** has no **:dimension** keyword; only scans using send address order are presently supported. Also, **segment-set-scan!!** has no **:include-self** keyword; in a **segment-set-scan!!** operation each processor always receives the result of applying the scan operation to all processors in its segment, including itself.

The *pvar* argument may be any pvar acceptable to the function specified as the *function* argument.

The *function* may be one of the following associative binary parallel functions:

+!!, and!!, or!!, max!!, min!!, copy!!, logand!!, logior!!, logxor!!

The *segment-set-pvar* must be a segment set pvar, as returned by the function **create-segment-set!!**. (See the dictionary entry of **create-segment-set!!** for more information.)

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.

REFERENCES

See also these related segment set operators:

segment–set–end–bits	**segment–set–end–bits!!**
segment–set–end–address	**segment–set–end–address!!**
segment–set–start–bits	**segment–set–start–bits!!**
segment–set–start–address	**segment–set–start–address!!**

segment–set–processor–not–in–any–segment
segment–set–processor–not–in–any–segment!!

self!! *[Function]*

Returns an address-object pvar containing the NEWS (grid) coordinates of each processor.

SYNTAX

self!!

ARGUMENTS

Takes no arguments.

RETURNED VALUE

address–object–pvar
: Temporary address-object pvar. In each active processor, contains an address object representing the NEWS (grid) coordinates of that processor.

SIDE EFFECTS

The returned value is allocated on the stack.

DESCRIPTION

This function returns an address object pvar 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*)**.

REFERENCES

See also the related functions

enumerate!!	**rank!!**	
self–address!!	**self–address–grid!!**	**sort!!**

See also the related operations

address–nth	**address–nth!!**
address–plus	**address–plus!!**
address–plus–nth	**address–plus–nth!!**
address–rank	**address–rank!!**
grid	**grid!!**
grid–relative!!	

self–address!! [*Function*]

Returns a pvar containing, in each processor, the send address of that processor.

SYNTAX

self–address!!

ARGUMENTS

Takes no arguments.

RETURNED VALUE

self–address–pvar Temporary integer pvar. In each active processor, contains the send address of that processor.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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

EXAMPLES

An example of a call to **self–address!!** from top level is the expression

```
(ppp (self-address!!) :end 10)
```

which displays the following:

```
0 1 2 3 4 5 6 7 8 9
```

The **self–address!!** function is most commonly used in combination with processor selection operators to select a specific subset of processors. For example,

```
(ppp (if!!     (evenp!! (self-address!!))
               (!! 0)
               (!! 1)) :end 10)
0 1 0 1 0 1 0 1 0 1
```

More complex selections of processors can be specified by combining the **self–address!!** function with mathematical operators such as **mod!!**.

```
(*defvar mod-pvar)
(*set mod-pvar (mod!! (self-address!!) (!! 4)))

(ppp mod-pvar :end 14)
0 1 2 3 0 1 2 3 0 1 2 3 0 1

(ppp (if!! (<!! mod-pvar (!! 2))
           (!! 1)
           (!! 0))
     :end 14)
1 1 0 0 1 1 0 0 1 1 0 0 1 1
```

REFERENCES

See also these related operations:

enumerate!! **rank!!** **self!!**
self–address–grid!! **sort!!**

See also these related send and grid address translation operators:

cube–from–grid–address **cube–from–grid–address!!**
cube–from–vp–grid–address **cube–from–vp–grid–address!!**
grid–from–cube–address **grid–from–cube–address!!**
grid–from–vp–cube–address **grid–from–vp–cube–address!!**

self–address–grid!! [*Function*]

Returns a pvar containing in each processor the grid (NEWS) coordinate of that processor along a specified dimension.

SYNTAX

self–address–grid!! *dimension–pvar*

ARGUMENTS

dimension–pvar	Integer pvar. Dimension for which the grid (NEWS) coordinate of the corresponding processor is determined.

RETURNED VALUE

coord–pvar	Temporary integer pvar. In each active processor, contains the grid (NEWS) coordinate of that processor along the dimension specified by *dimension–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function returns a pvar that contains the coordinate, along the dimension specified by *dimension–pvar*, of each selected processor.

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 VP set.

EXAMPLES

Assuming a two-dimensional grid, the expression

```
(ppp (self-address-grid!! (!! 0)) :mode :grid :end '(4 4))
```

displays the values

```
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3
```

and the expression

```
(ppp (self-address-grid!! (!! 1)) :mode :grid :end '(4 4))
```

displays the values

```
0 0 0 0
1 1 1 1
2 2 2 2
3 3 3 3
```

The following code fragment selects the diagonal elements of the grid,

```
(*when (=!! (self-address-grid!! (!! 0))
            (self-address-grid!! (!! 1)))
  ...
  )
```

and the following fragment selects the tridiagonal elements of the grid:

```
(*when (or!! (=!! (self-address-grid!! (!! 0))
                  (self-address-grid!! (!! 1)))
             (=!! (self-address-grid!! (!! 0))
                  (1+!! (self-address-grid!! (!! 1))))
             (=!! (self-address-grid!! (!! 0))
                  (1-!! (self-address-grid!! (!! 1)))))
  ...
  )
```

NOTES

Language Note:

A processor's grid address is distinct from its send address, even on a one-dimensional grid, and there is no guarantee that the two will be the same under any circumstances.

For example, assuming a one-dimensional grid has been defined, the following results might be obtained:

```
(ppp (self-address!!) :end 12)

0 1 2 3 4 5 6 7 8 9 10 11

(ppp (self-address-grid!! (!! 0)) :end 12)

3 2 0 1 5 4 6 7 256 255 253 254
```

Performance Note:

The computation of a grid self address using **self–address–grid!!** takes a signficant amount of time. Rather than calling **self–address–grid!!** over and over again, it is preferable to call it once. For example, the tridiagonal element selection example above may be more efficiently written as

```
(*let ((x-addr (self-address-grid!! (!! 0)))
       (y-addr (self-address-grid!! (!! 1))))
  (declare (type (field-pvar *current-send-address-length*)
                 x-addr y-addr))
  (*when (or!! (=!! x-addr y-addr)
               (=!! x-addr (1+!! y-addr))
               (=!! x-addr (1-!! y-addr)))
    ...
    ))
```

REFERENCES

See also these related operations:

enumerate!! **rank!!** **self!!**
self–address!! **sort!!**

See also these related send and grid address translation operators:

cube–from–grid–address **cube–from–grid–address!!**
cube–from–vp–grid–address **cube–from–vp–grid–address!!**
grid–from–cube–address **grid–from–cube–address!!**
grid–from–vp–cube–address **grid–from–vp–cube–address!!**

*set [*Macro*]

Copies the supplied source pvars into the supplied destination pvars.

SYNTAX

***set** *destination–pvar–1 source–pvar–1*
destination–pvar–2 source–pvar–2
...
destination–pvar–n source–pvar–n

ARGUMENTS

destination–pvar	Pvar expression. Pvar into which values are copied. Must evaluate to a non-temporary pvar.
source–pvar	Pvar expression. Pvar from which values are copied. May evaluate to any pvar.

RETURNED VALUE

nil	Evaluated for side effect only.

SIDE EFFECTS

The macro ***set** evaluates each pair of *source–pvar* and *destination–pvar* arguments in order. In all active processors, the value of *source–pvar* is copied into the pvar obtained by evaluating *destination–pvar*.

DESCRIPTION

This macro sets the contents of *destination–pvar* to the contents of *source–pvar* in all processors of the currently selected set, for each pair of *source–pvar* and *destination–pvar* arguments. Note that both *source–pvar* and *destination–pvar* are evaluated.

It is an error to attempt to ***set** the value of a temporary pvar. Temporary pvars are returned by *Lisp functions such as **!!** and **+!!**. The *Lisp simulator catches this error and prints an error message. Neither the *Lisp interpreter nor the *Lisp compiler catches this error.

EXAMPLES

The following examples show how ***set** may be used to copy values between pvars:

```
(*defvar pvar1 (!! 2))
(*defvar pvar2 (self-address!!))
(*defvar dest)

;;; set dest to product of pvar1 and pvar2 in each processor
(*set dest (*!! pvar1 pvar2))

(ppp dest :end 8)
0 2 4 6 8 10 12 14

;;; set dest to the value of pvar1 in each processor
;;; where the value of pvar2 is less than 4
(*when (<!! pvar2 (!! 4))
    (*set dest pvar1))

(ppp dest :end 8)
2 2 2 2 8 10 12 14
```

As an example of how *not* to use ***set**, consider the function **foo** below.

```
(defun foo (x) (*set x (!! 5)))
```

These calls to the function **foo** violate the rule against setting the value of a temporary pvar, and are therefore in error:

```
(foo (!! 3))
(foo (cos!! (+!! a b)))
```

To modify array elements and structure pvar slots, use the ***setf** macro. See the dictionary entry for ***setf** for more information.

NOTES

Important:

The ***set** macro evaluates its first argument, as does the Common Lisp **set** operator. The values contained in this argument, which must be a permanent, global, or local pvar, are destructively modified.

*setf [Function]

Destructively modifies the pvars specified by the supplied accessor functions to contain the values specified by the supplied pvar expressions.

SYNTAX

***setf** *pvar–accessor–1 pvar–expression–1*
pvar–accessor–2 pvar–expression–2
...
pvar–accessor–n pvar–expression–n

ARGUMENTS

pvar–accessor	*Lisp pvar accessor expression. Indicates pvar to be modified.
pvar–expression	Pvar expression. Value to be stored at the specified location.

RETURNED VALUE

nil	Evaluated for side effect only.

SIDE EFFECTS

Destructively modifies the location specified by *pvar–accessor* to contain the value of *pvar–expression*, for each pair of *pvar–accessor* and *pvar–expression.*

DESCRIPTION

This is the *Lisp equivalent of the Common Lisp **setf** macro. This operation takes one or more sets of *pvar–accessor* and *pvar–expression* pairs. It evaluates the *pvar–expression* of each pair, and converts the *pvar–accessor* to an expression that modifies the specified location. For each pair, the location referenced by the *pvar–accessor* is modified to contain the value of *pvar–expression.* The ***setf** macro must be used—and the Common Lisp **setf** must not be used—to modify locations referenced by pvar accessor expressions.

Each *pvar–accessor* must be one of:

a symbol whose value is a pvar, in which case the ***setf** call behaves like a call to ***set**.

a call to one of the operators

aref!!	**sideways–aref!!**
row–major–aref!!	**row–major–sideways–aref!!**
pref	**pref!!**
realpart!!	**imagpart!!**
load–byte!!	**ldb!!**

a call to a structure slot accessor defined by ***defstruct**

a call to a function for which an appropriate modifier has been defined by the use of ***defsetf**.

an expression of the form **(the** *data–type pvar–accessor***)**, where *pvar–accessor* is one of the possible forms listed above

EXAMPLES

The operation performed by ***setf** depends on the type of *pvar–accessor* to which it is applied. For example, a call to ***setf** such as

```
(*setf (pref int-pvar 387) 15)
```

changes the value of **int–pvar** in processor 387 to 15.

The most common use of ***setf** is to change the value of pvar array elements and pvar structure slots. 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 **slot1** value of the structure pvar **foo–struct–pvar** to 84 in each processor.

Accessor forms can be nested, as in the expression

```
(*setf (pref (aref!! array-pvar (!! 3)) 29) 100)
```

which changes the value of element 3 of **array-pvar** in processor 29 to 100. Not all nestings of operators work, however. For example, the expression

```
(*setf (aref (pref array-pvar 29) 3) 100)
```

will not perform the same operation as the above example, because the operator **aref** is not one of the parallel accessors that ***setf** recognizes.

NOTES

Using ***setf** to modify the **realpart!!** and **imagpart!!** parts of a complex pvar is a *Lisp extension; there is no corresponding functionality in Common Lisp (that is, you can't **setf** the **realpart** or **imagpart** of a scalar complex value).

Usage Note:

The ***setf** macro implicitly performs an **alias!!** operation on array pvar references and parallel structure slot accessor forms. (See the entry for the **alias!!** macro.) It is therefore unnecessary to explicitly enclose these types of arguments in calls to **alias!!**. For example, the **alias!!** is unnecessary in the expression:

```
(*setf (alias!! (aref!! array-pvar (!! 3))) (!! 29))
```

Performance Notes:

Applying ***setf** to a parallel array reference with nonconstant indices, as in

```
(*setf (aref!! array-pvar (random!! (!! 6)))  (!! 4))
```

is permitted in the CM-2 implementation of *Lisp, but is relatively inefficient compared with applying ***setf** to references with constant indices, such as

```
(*setf (aref!! array-pvar (!! 6))  (!! 4))
```

On the other hand, using ***setf** on sideways arrays with non-constant indices is an efficient operation. (See the definitions of ***sideways-array** and **sideways-aref!!** for more information.)

Also, applying ***setf** to **pref!!** is equivalent to a call to ***pset**:

```
(*setf (pref!! dest-pvar address-pvar) source-pvar)
    <=>
(*pset :no-collisions source-pvar dest-pvar address-pvar)
```

Calling ***pset** directly is preferred as being more stylistically correct, as these two forms are functionally equivalent and the latter is somewhat more readable.

REFERENCES

See also these related operations:

***defsetf**	***set**	***undefsetf**

set–char–bit!! [*Function*]

Sets the state of a single flag bit of the supplied character pvar.

The returned pvar is allocated on the stack.

SYNTAX

set–char–bit!! *character–pvar bit–name–pvar newvalue–pvar*

ARGUMENTS

character–pvar	Character pvar. Pvar for which bit selected by *bit–name–pvar* is set.
bit–name–pvar	Integer pvar. Selects bit to be tested in each active processor. Must contain integers in the range 0 to 3 inclusive.
newvalue–pvar	Boolean pvar. State (set/cleared) to which specified bit is set.

RETURNED VALUE

new–char–pvar	Temporary character pvar. In each active processor, contains a copy of the character in *character–pvar* with the flag bit specified by *bit–name–pvar* set to the value specified by *newvalue–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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!!**.

EXAMPLES

```
(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)
```

NOTES

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)

REFERENCES

See also the related character pvar attribute operators:

char–bit!! **char–bits!!** **char–code!!**
char–font!! **initialize–character** **set–char–bit!!**

set–vp–set *[Function]*

Make the specified VP set the current VP set.

SYNTAX

set–vp–set *vp–set*

ARGUMENTS

vp–set	VP set object. VP set to be made current. Must be defined, and must be allocated if voidable.

RETURNED VALUE

vp–set	The supplied *vp–set* argument is returned.

SIDE EFFECTS

Sets the value of ***current–vp–set*** to *vp–set*.

DESCRIPTION

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, in the case of flexible VP sets, instantiated.

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

REFERENCES

See also the following VP set operators:

create–vp–set	**deallocate–def–vp–sets**
deallocate–vp–set	**def–vp–set**
let–vp–set	***with–vp–set**

set–vp–set–geometry [*Function*]

Modifies the geometry of a VP set.

SYNTAX

set–vp–set–geometry *vp–set geometry–obj*

ARGUMENTS

vp–set	VP set for which the geometry is altered.
geometry–obj	Geometry object, as returned by **create–geometry**. Defines new geometry of *vp–set*.

RETURNED VALUE

nil	Evaluated for side effect only.

SIDE EFFECTS

The geometry of *vp–set* is altered to the dimensions specified by *geometry–obj*.

DESCRIPTION

Modifies the geometry of the specified *vp–set*, rearranging its values into the configuration specified by *geometry–obj*. The *vp–set* argument must be a defined and instantiated VP set.

The parameter *geometry–obj* must be a geometry object created with **create–geometry**, and the number of processors it specifies must match the total number of processors in *vp–set*.

Important: The **set–vp–set–geometry** operation only changes the *arrangement* of processors in a VP set, not the total number of processors. The effect of supplying a *geometry–obj* that would change the total number of VP's in the *vp–set* is undefined.

EXAMPLES

```
(setq geometry-1 (create-geometry :dimensions '(256 256)))
(setq geometry-2 (create-geometry :dimensions '(65536)))

(setq vp-set-1 (create-vp-set nil :geometry geometry-1))

(set-vp-set-geometry vp-set-1 geometry-2)
```

REFERENCES

See also the following flexible VP set operators:

allocate-vp-set-processors **allocate-processors-for-vp-set**
deallocate-vp-set-processors **deallocate-processors-for-vp-set**
with-processors-allocated-for-vp-set

See also the following geometry definition operator:

create-geometry

See also the following VP set definition and deallocation operators:

def-vp-set **create-vp-set** **let-vp-set**

sideways–aref!! [*Function*]

Performs a parallel array reference on the supplied sideways array pvar.

SYNTAX

sideways–aref!! *array–pvar* **&rest** *subscript–pvars*

ARGUMENTS

array–pvar	Array pvar from which values are referenced. Must have been turned sideways by ***sideways–array** or ***slicewise**.
subscript–pvars	Integer pvars. Specify array element to be referenced in each processor.

RETURNED VALUE

value–pvar	Temporary pvar. Value retrieved in each processor.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function performs a parallel array reference, similar to **aref!!**, on an array that has been turned sideways by ***sideways–array** or ***slicewise**. In general, especially for large arrays, non-constant indexing can be very slow. Turning arrays sideways allows the CM-2 architecture to do non-constant indexing in constant time. However, sideways arrays can only be referenced by using **sideways–aref!!**.

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.

EXAMPLES

These expressions declare and define an array pvar that can be turned sideways. In each processor, the array [5.0 8.0] is stored.
[3.0 0.0]

```
(*proclaim '(type (array-pvar single-float '(2 2))
                  my-sideways-array))

(*defvar my-sideways-array (!! #2A((5.0 8.0) (3.0 0.0))))
```

The array is turned sideways, and is verified to be slicewise.

```
(*sideways-array my-sideways-array)

(sideways-array-p my-sideways-array)  =>  T
```

The following expression defines two pvars containing non-constant indices, and then uses **sideways-aref!!** to perform a parallel array reference on the array pvar **my-sideways-array**.

```
(*let ((index-1  (mod!! (self-address!!) (!! 2))))
       (index-2  (mod!! (floor!! (self-address!!) (!! 2))
                        (!! 2))))
   (ppp (sideways-aref!! my-sideways-array
                        index-2 index-1)
        :end 14))

5.0 8.0 3.0 0.0 5.0 8.0 3.0 0.0 5.0 8.0 3.0 0.0 5.0 8.0
```

The above example uses **mod!!** for clarity. It can also be written as:

```
(*let ((index-1 (load-byte!! (self-address!!) (!! 0) (!! 1)))
       (index-2 (load-byte!! (self-address!!) (!! 1) (!! 1))))
    (ppp (sideways-aref!! my-sideways-array
                          index-2 index-1)
         :end 14))
```

The **sideways-aref!!** function may also be used with ***setf** to modify the values stored in a sideways array. For example, given the following declarations

```
(*proclaim '(type (array-pvar single-float '(2))
                  my-sideways-array))
(*defvar my-sideways-array (!! #(5.0 0.0)))
```

this example demonstrates the use of ***setf** to store values into an array pvar using constant indices:

```
(*setf (sideways-aref!! my-sideways-array (!! 1)) (!! 6.0))

(ppp (sideways-aref!! my-sideways-array
                     (mod!! (self-address!!) (!! 2)))
     :end 14)
5.0 6.0 5.0 6.0 5.0 6.0 5.0 6.0 5.0 6.0 5.0 6.0 5.0 6.0
```

and this example shows the use of ***setf** with non-constant indices.

```
(*setf (sideways-aref!! my-sideways-array
                      (mod!! (self-address!!) (!! 2)))
       (!! 7.0))

(ppp (sideways-aref!! my-sideways-array (!! 0)) :end 14)
7.0 6.0 7.0 6.0 7.0 6.0 7.0 6.0 7.0 6.0 7.0 6.0 7.0 6.0

(ppp (sideways-aref!! my-sideways-array (!! 1)) :end 14)
5.0 7.0 5.0 7.0 5.0 7.0 5.0 7.0 5.0 7.0 5.0 7.0 5.0 7.0
```

Note that the result of the second example depends on the result of the first.

NOTES

The **sideways-aref!!** function works in the same way as **aref!!** does except that 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 being sideways.

There are some important restrictions on the size of arrays passed as arguments to **sideways-aref!!**. The *array-pvar* argument must be an array pvar that has been turned sideways. Arrays that have been turned sideways must contain elements whose lengths are powers of 2 or multiples of 32. Further, the total number of bits the sideways array occupies in CM memory must be divisible by 32. This number can be determined either by **(pvar-length array-pvar)** or by multiplying the total number of elements in the array by the size of an individual element.

REFERENCES

See also the related array-referencing operations:

aref!! **row-major-aref!!**
row-major-sideways-aref!!

The following operations convert arrays to and from sideways orientation:

***processorwise** ***sideways-array** ***slicewise**

See the definition of the ***sideways-array** operation for more information about sideways arrays.

*sideways–array [*Defun]

Toggles an array between processorwise and sideways (slicewise) orientations.

SYNTAX

***sideways–array** *array–pvar*

ARGUMENTS

array–pvar	Array pvar to be converted.

RETURNED VALUE

t	Evaluated for side effect only.

SIDE EFFECTS

Converts *array–pvar* to sideways orientation if it is in normal orientation. Converts *array–pvar* back to normal orientation if it is in sideways orientation.

DESCRIPTION

The function ***sideways–array** forces *array–pvar* to be addressed in a sideways (slicewise) ordering. Calling ***sideways–array** on an array that is already sideways returns it to a processorwise ordering.

EXAMPLES

The following example shows how one might use slicewise arrays. Given the vector pvar defined by

```
(*proclaim '(type (vector-pvar single-float 20)
              my-sideways-vector))
(*defvar my-sideways-vector
         (make-array!! 20 :element-type 'single-float-pvar))
```

the following code example calls a user-defined function to fill **my–sideways–vector** with data, uses ***sideways–array** to turn it sideways so that it can be accessed using indirect addressing, calls another user-defined function to operate on the sideways vector pvar, and finally uses ***sideways–array** again to return it to processorwise orientation, so that its values can be accessed and displayed.

```
(defun main ()
   (fill-my-sideways-vector-with-values)
   (*sideways-array my-sideways-vector)
   (do-computations-on-my-sideways-vector)
   (*sideways-array)
   (ppp my-sideways-vector :end 10))
```

NOTES

Implementation Note:

Turning an array sideways (slicewise) allows the CM-2 hardware to more efficiently reference arrays using indirect addressing. On the CM-2, indirect addressing is array referencing in which a different array element is accessed in each processor.

Usage Notes:

There are some important restrictions on the size of arrays passed as arguments to ***sideways–array**. These restrictions extend to the related functions ***processorwise** and ***slicewise.**

The *array–pvar* argument must be an array pvar that contains elements whose lengths are powers of 2 or multiples of 32. Further, the total number of bits the array occupies in CM memory must be divisible by 32. This number can be determined either by **(pvar–length array–pvar)** or by multiplying the total number of elements in the array by the size of an individual element.

The ***sideways–array** function is most efficient when the array elements of *array–pvar* are each 32 bits long.

REFERENCES

See also the functions ***processorwise, sideways–aref!!, sideways–array–p**, and ***slicewise.**

sideways–array–p *[Function]*

Tests whether the supplied array is currently in sideways (slicewise) orientation.

SYNTAX

sideways–array–p *array–pvar*

ARGUMENTS

array–pvar	Array pvar. Pvar to be tested for sideways orientation.

RETURNED VALUE

sideways–array–p	Boolean. The value **t** if *array–pvar* is in sideways (slicewise) orientation, and the value **nil** if it is in normal orientation.

SIDE EFFECTS

None.

DESCRIPTION

Tests the specified array pvar, returning **t** if it is sideways (slicewise) and **nil** otherwise.

Turning an array sideways, via one of the functions ***sideways–array**, ***slicewise**, or ***processorwise**, allows special Connection Machine hardware to more efficiently reference arrays using indirect addressing. On the CM, indirect addressing is array referencing in which a different array element is accessed in each processor.

REFERENCES

For more information on giving an array pvar a sideways orientation, see the dictionary entries for ***processorwise**, ***sideways–array**, and ***slicewise**.

---

signum!! [Function]

Returns a pvar indicating the sign of the supplied pvar.

SYNTAX

signum!! *numeric–pvar*

ARGUMENTS

numeric–pvar	Numeric pvar. Pvar for which sign is determined.

RETURNED VALUE

signum–pvar	Temporary pvar, of same type as *numeric–pvar*. In each active processor, contains the signum of the value of *numeric–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function returns a pvar containing the signum of the values of the *numeric-pvar* argument. This is defined as follows:

For integer and floating-point values, this function returns –1, 0, or 1 in each processor according to whether the value of *numeric–pvar* in that processor is negative, zero, or positive. For floating-point pvars, the result is a floating-point pvar of the same format as the *numeric–pvar* argument.

For complex pvars, this function returns in each processor either the unit-length complex value that has the same phase as the value of *numeric–pvar*, or complex zero, if *numeric–pvar* contains a complex zero.

sin!!, sinh!! [*Function*]

Takes the sine and hyperbolic sine of the supplied pvar.

SYNTAX

sin!! *radians–pvar*
sinh!! *radians–pvar*

ARGUMENTS

radians–pvar Numeric pvar. Angle, in radians, for which the sine (hyperbolic sine) is calculated.

RETURNED VALUE

result–pvar Temporary numeric pvar. In each active processor, contains the sine (hyperbolic sine) of *radians–pvar.*

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The function **sin!!** returns the sine of *radians–pvar*.

The function **sinh!!** returns the hyperbolic sine of *radians–pvar*.

---

*slicewise [*Defun]

Converts a normal, processorwise array to sideways (slicewise) orientation.

SYNTAX

***slicewise** *array–pvar*

ARGUMENTS

array–pvar	Array pvar. Normal orientation array pvar to be converted.

RETURNED VALUE

t	Evaluated for side effect only.

SIDE EFFECTS

Converts *array–pvar* from normal orientation to sideways orientation.

DESCRIPTION

Converts a normal, processorwise array to slicewise (sideways) orientation. An error is signalled if the array is not in processorwise orientation. Turning an array sideways allows the CM to efficiently get array values using indirect addressing (array references in which a different array element is accessed in each processor).

The *array–pvar* argument must contain elements with lengths that are powers of 2 or multiples of 32, and the **pvar–length** of the array must be divisible by 32. The ***slicewise** function is most efficient when the array elements of *array–pvar* are each 32 bits long.

REFERENCES

See also the functions ***processorwise**, ***sideways–array**, and **sideways–array–p**.

some!! *[Function]*

Tests in parallel whether the supplied pvar predicate is true for at least one set of elements having the same indices in the supplied sequence pvars.

SYNTAX

some!! *predicate sequence–pvar* **&rest** *sequence–pvars*

ARGUMENTS

predicate Boolean pvar predicate. Used to test elements of sequences in the *sequence–pvar* arguments. Must take as many arguments as the number of *sequence–pvar* arguments supplied.

sequence–pvar, *sequence–pvars*
Sequence pvars. Pvars containing, in each processor, sequences to be tested by *predicate*.

RETURNED VALUE

some–pvar Temporary boolean pvar. Contains the value **t** in each active processor in which at least one set of elements having the same indices in the sequences of the *sequence–pvars* satisfies the *predicate*. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The **some!!** function returns a boolean pvar indicating in each processor whether the supplied *predicate* is true for at least one set of elements with the same indices in the sequences of the supplied *sequence–pvars*.

In each processor, the *predicate* is first applied to the index 0 elements of the sequences in the *sequence–pvars*, then to the index 1 elements, and so on. The *n*th time *predicate*

is called, it is applied to the *nth* element of each of the sequences. If *predicate* returns **t** in any processor, that processor is temporarily removed from the currently selected set for the remainder of the operation. The operation continues until the shortest of the *sequence–pvars* is exhausted, or until no processors remain selected.

The pvar returned by **some!!** contains **t** in each processor where *predicate* returns the value **t** for at least one set of sequence elements. If *predicate* returns **nil** for every set of sequence elements in a given processor, **some!!** returns **nil** in that processor.

EXAMPLES

```
(some!! 'equalp!! (!! #(1 2 3)) (!! #(1 2 3)))   <=>  t!!
(some!! 'equalp!! (!! #(1 2 3)) (!! #(1 2 6)))   <=>  t!!
(some!! 'equalp!! (!! #(1 2 3)) (!! #(1 2 3 4))) <=>  t!!
(some!! 'equalp!! (!! #(1 2 3)) (!! #(2 6 9)))   <=>  nil!!
```

NOTES

Compiler Note:

The *Lisp compiler does not compile this operation.

REFERENCES

See the related functions **every!!**, **notany!!**, and **notevery!!**.

See also the general mapping function **amap!!**.

sort!! [Function]

Performs a parallel sort on the values of the supplied pvar.

SYNTAX

sort!! *pvar predicate* **&key :dimension :segment–pvar :key**

ARGUMENTS

numeric–pvar	Non-complex numeric pvar. Pvar containing values to be sorted.
predicate	Two-argument pvar predicate. Determines type of sort. Currently limited by implementation to the function **<=!!**.
:dimension	Integer or **nil**. Specifies dimension along which to perform ranking. The default, **nil**, specifies a send-address order ranking. If not **nil**, this argument must be an integer between 0 inclusive and ***number–of–dimensions*** exclusive.
:segment–pvar	Segment pvar or **nil**. Specifies segments in which to perform sort. The default, **nil**, specifies an unsegmented sort.
:key	One-argument pvar function. Applied to *pvar* before sort is performed.

RETURNED VALUE

sort–pvar	Temporary pvar. In all active processors, contains the values of *pvar* sorted into the order specified by *predicate*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

In all active processors, **sort!!** sorts the values of the supplied *pvar*.

The keywords, **:dimension** and **:segment-pvar** permit rankings to be taken along specific grid dimensions and within segments.

The **:dimension** keyword specifies whether the sorting is done by send address order or along a specific dimension. If a dimension is specified, sorting is performed only along that dimension. The default value, **nil**, specifies a send-address order sort.

For example, assuming a two-dimensional grid, a **:dimension** argument of 0 causes sorting to occur independently in each "row" of processors along dimension 0. A **:dimension** argument of 1 causes sorting to occur independently in each "column" of processors along dimension 1 (see Figure 4).

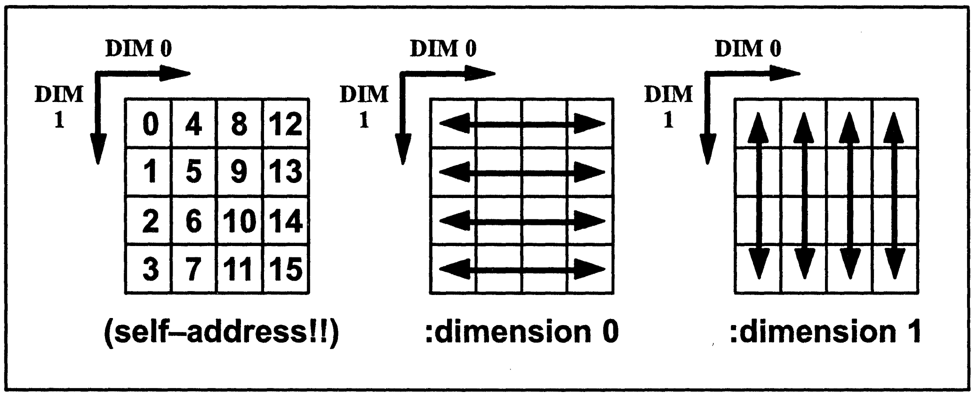

Figure 4. Effect of different **:dimension** arguments, assuming a two-dimensional grid.

The **:segment-pvar** argument specifies whether sorting is performed separately within segments. The default is **nil**; **sort!!** is by default unsegmented. If provided, the **:segment-pvar** value must be a segment pvar. A segment pvar contains boolean values, **t** in the first processor of each segment and **nil** in all other processors. If a segment pvar is specified, then sorting is done independently within each segment.

If both a **:dimension** and a **:segment pvar** argument are specified, then the sort is done independently for each "row" along the specified dimension and independently within segments for each row.

The **:key** argument allows selection of a key on which the sort is done. For instance, a ***defstruct** (parallel structure) slot accessor function could be provided as the **:key** argument and a pvar of the associated ***defstruct** type could be supplied as the *pvar* argument. A **sort!!** with these arguments would sort the values of the supplied *pvar* based on the value of the accessed slot in each processor.

EXAMPLES

A sample call to **sort!!** is

```
(sort!! numeric-pvar)
```

Assume that **numeric–pvar** contains the following values, with * standing for an unselected processor:

```
7  *  2  3  *  1  0  6  .  .  .
```

Assuming that all other active processors contain values greater than those shown here, the result of the above call to **sort!!** is a pvar containing the values

```
0  *  1  2  *  3  6  7 . . .
```

Notice that data in unselected processors remains unchanged.

A sample call to **sort!!** with a **:segment–pvar** argument is

```
(sort!! data-pvar '<=!!
        :segment-pvar (evenp!! (self-address!!)))
```

If **data–pvar** contains the values

0 2 4 2 1 7 5 3 4 7 8 2...

then (again assuming that all other processors contain larger values than those shown here) the returned pvar would contain the values

0 2 2 4 1 7 3 5 4 7 2 8...

An example of **sort!!** with a **:dimension** argument is

```
(sort!! data-pvar '<=!! :dimension 1)
```

Assuming the two-dimensional VP set geometry defined by

```
(*cold-boot :initial-dimensions '(4 4))
```

if the expression

```
(ppp data-pvar :mode :grid)
```

displays the values

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

then the expression

```
(ppp (sort!! (self-address!!) '<=!! :dimension 1) :mode :grid)
```

will display the values

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

A sample call to **sort!!** with a **:key** argument is

```
(sort!! foo '<=!! :dimension 0 :key 'foo-a!!)
```

If **foo** is an instance of a ***defstruct** parallel structure with a slot named **foo–a!!**, then this expression sorts **foo** based on the value of the **a** slot in each processor. Also, because the **:dimension** argument is 0, the sort takes place independently for each coordinate along dimension 0.

NOTES

The sort performed by **sort!!** is not guaranteed to be stable. If *numeric–pvar* contains the same value in two or more active processors, the order in which these values are returned in *rank–pvar* is arbitrary and indeterminate.

Compiler Note:

The *Lisp compiler does not compile **sort!!** if a **:segment–pvar** argument is supplied.

REFERENCES

See also the related functions

enumerate!!	**rank!!**	**self!!**
self–address!!	**self–address–grid!!**	

spread!! [*Function*]

Spreads values of a pvar from one coordinate of a grid dimension to all coordinates along that dimension.

SYNTAX

spread!! *pvar dimension coordinate*

ARGUMENTS

pvar	Pvar expression. Pvar containing values to be spread.
dimension	Integer or **nil**. Index, zero-based, of dimension along which values are spread. If **nil**, a send-address order spread is performed, and *coordinate* specifies a send address.
coordinate	Integer. Coordinate along *dimension* from which to spread values.

RETURNED VALUE

spread–pvar	Temporary pvar, of same type as *pvar*. Contains the result of spreading the values of *pvar* from the specified *coordinate* of the grid dimension specified by *dimension* to all processors along the length of the dimension.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function spreads data across the Connection Machine processors along dimension *dimension*. The data is taken from the processor at the specified *coordinate* and spread to all active processors along the specified *dimension*. (See Figure 5.)

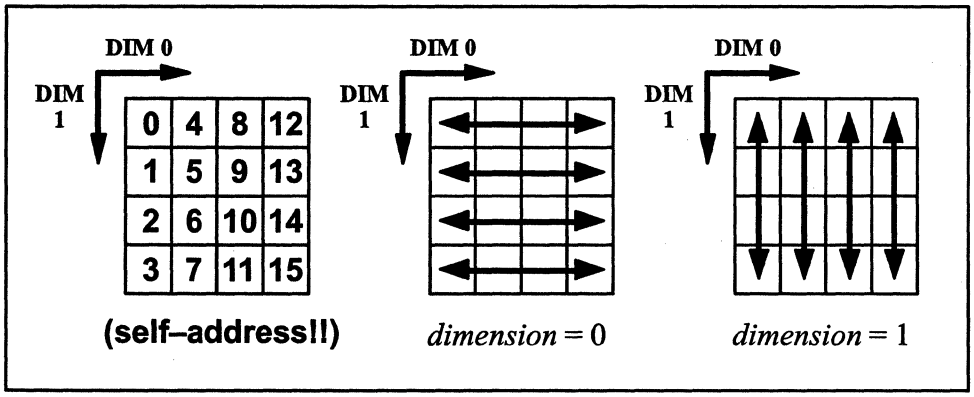

Figure 5. Effect of different *dimension* arguments, assuming a two-dimensional grid.

It is an error if *coordinate* specifies any processors that are not in the currently selected set.

EXAMPLES

Assuming a two-dimensional grid, and a pvar, **numeric-pvar**, containing the values

```
 1   2   3   4   5
 6   7   8   9  10
11  12  13  14  15
16  17  18  19  20
21  22  23  24  25
```

then the expression

```
(spread!! numeric-pvar 0 2)
```

returns a pvar containing the values

```
 3   3   3   3   3
 8   8   8   8   8
13  13  13  13  13
18  18  18  18  18
23  23  23  23  23
```

NOTES

Performance Note:

The expression

```
(!! (pref x 10))
```

can be used to spread data to all processors faster than the equivalent, but less efficient expression

```
(spread!! x nil 10)
```

which performs a send-address spread of data across all active processors.

REFERENCES

See also these related operations:

reduce–and–spread!! **scan!!** **segment–set–scan!!**

sqrt!! *[Function]*

Takes the square root of the supplied numeric pvar

SYNTAX

sqrt!! *numeric–pvar*

ARGUMENTS

numeric–pvar	Numeric pvar. Pvar for which the square root is calculated.

RETURNED VALUE

sqrt–pvar	Numeric pvar. In each active processor, contains the non-negative square root of the corresponding value of *numeric–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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

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

NOTES

The function **sqrt!!** will signal an error if its arguments are of one pvar type, yet contain values that would produce a result of another pvar type. For example, it is an error if *numeric–pvar* is either an integer or float pvar containing values less than zero in any processor. (This would produce a complex result in that processor.)

The reason **sqrt!!** is defined in this way is so that the pvar it returns can be guaranteed to be of a specific pvar type. If **sqrt!!** were allowed to return different data types in different processors, then it would have to return a general pvar as its result. Not only is this inefficient, it would also prevent **sqrt!!** expressions from compiling, because the *Lisp compiler does not compile expressions involving general pvars.

The general rule is that the **sqrt!!** function will not return a complex pvar as its result unless the supplied *numeric–pvar* argument is already a complex pvar or has been coerced to a complex pvar by use of **complex!!** or **coerce!!**:

```
(sqrt!! (coerce!! (!! -1) '(pvar (complex single-float))))
    <=>
(sqrt!! (complex!! (!! -1)))
    <=>
(!! #c(0.0 1.0))
```

standard–char–p!! [*Function*]

Performs a parallel test for standard characters on the supplied pvar.

SYNTAX

standard–char–p!! *character–pvar*

ARGUMENTS

character–pvar Character pvar. Tested in parallel for standard characters.

RETURNED VALUE

standard–charp–pvar

Temporary boolean pvar. Contains the value **t** in each active processor where the corresponding value of *character–pvar* is an character of type **standard–char**. Contains **nil** in all other processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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, i.e., a standard character is a character with zero bits and font attributes, that is defined as part of the Common Lisp standard character set.

---

string–char–p!! *[Function]*

Performs a parallel test for string characters on the supplied pvar.

SYNTAX

string–char–p!! *character–pvar*

ARGUMENTS

character–pvar Character pvar. Tested in parallel for string characters.

RETURNED VALUE

standard–charp–pvar

Temporary boolean pvar. Contains the value **t** in each active processor where the corresponding value of *character–pvar* is an character with bits and font attributes equal to zero. Contains **nil** in all other processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function returns **t** in those processors where character-pvar contains string-char data and **nil** in processors where *character–pvar* contains character data. Characters of type string-char have zero bits and font attributes.

REFERENCES

See also these related pvar data type predicates:

booleanp!!	**characterp!!**	**complexp!!**
floatp!!	**front–end–p!!**	**integerp!!**
numberp!!	**structurep!!**	**typep!!**

---

structurep!! *[Function]*

Tests whether the supplied pvar is a structure pvar.

SYNTAX

structurep!! *pvar*

ARGUMENTS

pvar	Pvar expression. Pvar to be tested.

RETURNED VALUE

boolean–pvar	A temporary pvar equal to **t!!** if *pvar* is a structure pvar, and **nil!!** otherwise.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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

REFERENCES

See also these related pvar data type predicates:

booleanp!!	**characterp!!**	**complexp!!**
floatp!!	**front–end–p!!**	**integerp!!**
numberp!!	**string–char–p!!**	**typep!!**

subseq!! *[Function]*

Extracts a subsequence in parallel from the supplied sequence pvar.

SYNTAX

subseq!! *sequence–pvar start* **&optional** *end*

ARGUMENTS

sequence–pvar	Sequence pvar. Pvar from which subsequence is extracted.
start	Integer pvar. Index, zero-based, of start of sequence to extract. Must contain identical values in all active processors.
end	Integer pvar. Index, zero-based, of end of sequence to extract. Must contain identical values in all active processors.

RETURNED VALUE

subseq–pvar	Sequence pvar. In each active processor, contains the subsequence of *sequence–pvar* specified by *start* and *end*.

SIDE EFFECTS

The returned value is allocated on the stack.

DESCRIPTION

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 uniform values in all active processors. Thus, the value of **(–!!** *end start***)** must be the same across all active processors.

EXAMPLES

```
(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
```

NOTES

Compiler Note:

The *Lisp compiler does not compile this operation.

REFERENCES

See also these related *Lisp sequence operators:

copy-seq!!	***fill**	**length!!**
***nreverse**	**reduce!!**	**reverse!!**

See also the generalized array mapping functions **amap!!** and ***map**.

substitute!!, substitute–if!!, substitute–if–not!!, [*Function*]

Performs a parallel substitution operation on the supplied sequence pvar, replacing specified old items with new items.

SYNTAX

substitute!!	*new–item old–item sequence–pvar*
	&key :test :test–not
	:start :end :count :from–end :key
substitute–if!!	*new–item test sequence–pvar*
	&key :start :end :count :from–end :key
substitute–if–not!!	*new–item test sequence–pvar*
	&key :start :end :count :from–end :key

ARGUMENTS

new–item	Pvar expression, of same data type as *sequence–pvar*. Item to substitute for *old–item* in each processor.
old–item	Pvar expression, of same data type as *sequence–pvar*. Item to be replaced in each processor.
test	One-argument pvar predicate. Test used in comparisons. Indicates a match by returning a non-**nil** result. Defaults to **eql!!**.
sequence–pvar	Sequence pvar. Pvar containing sequences to be modified.
:test	Two-argument pvar predicate. Test used in comparisons. Indicates a match by returning a non-**nil** result. Defaults to **eql!!**.
:test–not	Two-argument pvar predicate. Test used in comparisons. Indicates a match by returning a **nil** result.
:start	Integer pvar. Index of sequence element at which substitution starts in each processor. If not specified, search begins with first element. Zero-based.
:end	Integer pvar. Index of sequence element at which substitution ends in each processor. If not specified, search continues to end of sequence. Zero-based.

:count	Integer pvar. Maximum number of replacements to perform in each processor. Defaults to **(length!!** *sequence–pvar***)**
:from–end	Boolean. Whether to begin substitution from end of sequence in each processor. Defaults to **nil**.
:key	One-argument pvar accessor function. Applied to *sequence–pvar* before search is performed.

RETURNED VALUE

substitute–pvar	Temporary sequence pvar. In each active processor, contains a copy of the sequence from *sequence–pvar* with each element matching *old–item* replaced by *new–item*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

These functions are the parallel equivalent of the Common Lisp **substitute** functions.

In each processor, the **substitute!!** function searches *sequence–pvar* for elements that match *old–item*. The function returns a copy of *sequence–pvar* with each matching sequence element modified to contain the value specified by *new–item*. Elements of *sequence–pvar* are tested against *old–item* with the **eql!!** operator unless another comparison operator is supplied as either of the **:test** or **:test–not** arguments. The keywords **:test** and **:test–not** may not be used together. A lambda form that takes two pvar arguments and returns a boolean pvar result may be supplied as either the **:test** and **:test–not** argument.

In each processor, the function **substitute–if!!** searches *sequence–pvar* for elements satisfying *test*. The function returns a copy of *sequence–pvar* with each matching sequence element modified to contain the value specified by *new–item*. A lambda form that takes a single pvar argument and returns a boolean pvar result may be supplied as the *test* argument. Similarly, the function **substitute–if–not!!** searches *sequence–pvar* for elements failing *test*.

Arguments to the keywords **:start** and **:end** define a subsequence to be operated on in each processor.

The **:key** keyword accepts a user-defined function used to extract a search key from *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 **:count** keyword argument must be a positive integer pvar with values less than or equal to **(length!!** *sequence–pvar***)**. In each processor at most *count* elements are substituted.

NOTES

Compiler Note:

The *Lisp compiler does not compile this operation.

REFERENCES

This function is one of a group of similar sequence operators, listed below:

count!!	**count–if!!**	**count–if–not!!**
find!!	**find–if!!**	**find–if–not!!**
nsubstitute!!	**nsubstitute–if!!**	**nsubstitute–if–not!!**
position!!	**position–if!!**	**position–if–not!!**
substitute!!	**substitute–if!!**	**substitute–if–not!!**

See also the generalized array mapping functions **amap!!** and ***map**.

*sum [*Defun]

Returns the numeric sum of the values of a pvar.

SYNTAX

***sum** *numeric–pvar*

ARGUMENTS

numeric–pvar Numeric pvar. Pvar for which numeric sum is determined.

RETURNED VALUE

sum–of–values Scalar value. Numeric sum of the values of *numeric–pvar*.

SIDE EFFECTS

None.

DESCRIPTION

This returns a Lisp value that is the sum of the value of *numeric–pvar* in every selected processor. If there are no selected processors, ***sum** returns 0.

REFERENCES

See also the related global operators:

***and**	***integer–length**	***logand**
***logior**	***logxor**	***max**
***min**	***or**	***xor**

See also the related logical operators:

and!! **not!!** **or!!** **xor!!**

taken–as!! [*Function*]

Returns a copy of the supplied pvar interpreted as a pvar of the specified type.

SYNTAX

taken–as!! *pvar pvar–type* **&optional** *offset*

ARGUMENTS

pvar	Pvar expression. Pvar to be reinterpreted.
pvar–type	*Lisp type specifier. Pvar type into which *pvar* is reinterpreted.
offset	Integer. Offset in bits at which reinterpretation of *pvar* begins. Default is 0, indicating no offset.

RETURNED VALUE

taken–as–pvar	Temporary pvar of type specified by *pvar–type*. In each active processor, contains a copy of the value of *pvar* beginning at *offset*, considered as a value of type *pvar–type*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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 a temporary pvar containing the original bits of *pvar* interpreted as values in the data type *pvar–type*. No coercion or change in representation occurs. For example,

```
(taken-as!! (!! 1.0) '(pvar (unsigned-byte 32))
=> (!! 1065353216)
```

EXAMPLES

A sample call to **taken–as!!** is

```
(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)) unsigned8))
(*defvar unsigned8)
(fun-that-requires-unsigned-byte-8 unsigned8)
(fun-that-requires-bit-vector-8
  (taken-as!! unsigned8 '(pvar (bit-vector 8))))
(fun-that-requires-unsigned-byte-8 unsigned8)
```

Here, **unsigned8** is an unsigned-byte pvar of length 8. The call to **taken–as!!** allows **unsigned8** to be passed to a function that expects a bit-vector pvar of length 8.

The *offset* argument can be useful for selecting subportions of pvars. Consider the pvar **unsigned16** in this example:

```
(*proclaim '(type (pvar (unsigned-byte 16)) unsigned16))
(*defvar unsigned16)
(need-8 (taken-as!! unsigned16 '(pvar (unsigned-byte 8)) 4))
```

The pvar **unsigned16** is a 16-bit pvar. The function **need–8** requires an 8-bit pvar. Using **taken–as!!** on **unsigned16** with an *offset* argument of 4 extracts the 4th through the 11th bits of **unsigned16** in each processor to be treated as an **(unsigned–byte 8)** pvar.

NOTES

It is an error to specify a *pvar–type* and/or offset requiring more bits than are contained in *pvar*. It *is* legal, however, to specify a *pvar–type* that requires only a subset of the bits of *pvar*. This function relies on the internal representation of pvars in the Connection Machine system and therefore cannot work in the *Lisp simulator.

REFERENCES

See also the related *Lisp declaration operators: ***locally** ***proclaim** **unproclaim**
See also the related type coercion function **coerce!!**.

tan!!, tanh!! [*Function*]

Take the tangent and hyperbolic tangent of the supplied pvar.

SYNTAX

tan!! *radians–pvar*
tanh!! *radians–pvar*

ARGUMENTS

radians–pvar	Numeric pvar. Angle, in radians, for which the tangent (hyperbolic tangent) is calculated.

RETURNED VALUE

result–pvar	Temporary numeric pvar. In each active processor, contains the tangent (hyperbolic tangent) of *radians–pvar.*

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The function **tan!!** takes the tangent of its argument in each processor.
The function **tanh!!** takes the hyperbolic tangent of its argument in each processor.

---

*trace [*Macro*]

Enables tracing for the specified user-defined *Lisp functions.

SYNTAX

***trace &rest** **defun-function-names*

ARGUMENTS

**defun-function-names*
Symbols. Names of user-defined *Lisp functions to be traced.

RETURNED VALUE

traced-functions List of symbols. Names of functions traced.

SIDE EFFECTS

Enables tracing on the named functions. Has no effect on functions that are already traced.

DESCRIPTION

Enables tracing for the named parallel functions, which must have been defined using ***defun.**

EXAMPLES

Invoked at top level, **(*trace foo)** causes a message to be printed whenever the function **foo** is either called or exited. For example,

```
(*defun self-random!! ()
  (random!! (1+!! (self-address!!))))

(*trace self-random!!)  =>  (*DEFUN-SELF-RANDOM!!)

(self-random!!)  =>
1 Enter *DEFUN-SELF-RANDOM!!
1 Exit *DEFUN-SELF-RANDOM!! #<Structure PVAR A032B6>
#<Structure PVAR A03276>
```

A call **(*untrace self-random!!)** turns off this tracing mechanism.

```
(*untrace self-random!!)  =>  (*DEFUN-SELF-RANDOM!!)
```

REFERENCES

The macros ***trace** and ***untrace** are the parallel equivalents of the Common Lisp **trace** and **untrace** functions, defined in *Common Lisp: The Language*.

See also the following related operations:

***apply**	***defun**	***funcall**
un*defun		

trace–stack [*Function*]

Enables and disables tracing of CM stack usage by *Lisp programs.

SYNTAX

trace–stack &optional *trace–action verbose*

ARGUMENTS

trace–action	Type of trace to perform. May be any one of **:trace**, **:break**, **:error**, **:warn**, **:call**, **:status**, **:level**, **:max**, **:break–at–limit**, **:break–above–limit**, **:init**, **:reset**, **:newmax**, **:off**, or **nil**. Default value is **:trace**.
verbose	Boolean. Determines whether **trace–stack** displays status messages. Default value is **t**.

RETURNED VALUE

current–stack–level	Integer. Current level of the CM stack memory.
maximum–stack–limit	Integer. Maximum limit on stack usage. This is the current value of the *Lisp variable ***maximum–stack–level***.

SIDE EFFECTS

When tracing is enabled, this operator places an "advice" function around the internal Paris operator that allocates stack memory.

DESCRIPTION

The **trace–stack** operator is a tool that is used to trace CM stack usage of a *Lisp program. This typically involves a two step process:

- First, a stack trace is made of the program in which the maximum CM stack usage of the program is stored in the *Lisp variable ***maximum–stack–level***.

- Second, a trace is made of the execution of the program using the limit found by the first trace, such that whenever the program attempts to allocate stack memory at or beyond the traced limit, a break, error, or warning is signalled.

The **trace–stack** operator is used to select both of these tracing steps and to control a number of other trace-related features. The type of trace performed is determined by the *trace–action* argument, which defaults to **:trace**. The legal tracing options are:

:trace Turns on stack tracing, and sets ***maximum–stack–level*** to the current CM stack level. Every time the current stack usage meets or exceeds the value of ***maximum–stack–level***, the variable is updated to the new stack level.

:break Switches to *break* tracing. A continuable error is signalled whenever stack usage meets or exceeds the limit set by ***maximum–stack–level***.

:error Switches to *error* tracing. Same as **:break**, but a fatal error is signalled.

:warn Switches to *warning* tracing. Same as **:break**, but displays a warning.

You can also supply a *trace–action* argument of **:call**. This selects "function call" tracing, in which every time new CM memory is allocated, a **funcall** is made to the user-defined function specified by the *Lisp variable ***maximum–stack–function***. This function is passed two arguments: the current stack level and the value of ***maximum–stack–level***. This feature exists so that users can write their own stack-tracing operations.

The following operations are conveniences for the most common types of tracing:

:init Call **(*warm–boot)**, then turn on **:trace** stack tracing.

:reset Call **(*warm–boot)**, then switch to **:break** stack tracing.

:newmax Set ***maximum–stack–level*** to the current stack level.

A number of *trace–action* options simply display status information. These options are:

:status Display the current stack level and the ***maximum–stack–level***.

:level Displays just the current stack level.

:max Displays just the value of ***maximum–stack–level***.

Two of the *trace–action* options control the point at which a break/error is signalled:

:break–at–limit Signal when stack level reaches the current limit (the default).

:break–above–limit Signal only when stack level exceeds the current limit.

Finally, you can disable all stack tracing options by using either of the following options:

:off, nil Turn off stack tracing. (These two options are equivalent.)

EXAMPLES

The **trace–stack** function is designed to help you track the CM stack usage of your *Lisp programs. You'll find this function useful both when you want to determine the maximum amount of stack space that your program uses, and when you want to determine whether running your program with specific arguments causes it to exceed the "normal" amount of stack usage.

As a specific example, let's take the following simple function:

```
(defun test (a b c)
   (*!! a (+!! b c)))
```

We can run a simple stack trace of this function like this:

```
(*warm-boot) ;; To clean out the stack

(trace-stack)
Stack tracing is now on in :TRACE mode.
Current stack level is 1536.
Maximum stack limit is 1536.
1536
1536

(test 9 3 2)
#<FIELD-Pvar 9-7 *DEFAULT-VP-SET* (128 64)>
```

The maximum stack limit is now set to the amount of stack memory we used by calling the **test** function.

Now let's switch to the **:break** mode, and clear the stack again:

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

(*warm-boot) ;; Clean out stack again
```

We can call **trace-stack** to see what the current settings are:

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

Now let's repeat the call to the test function:

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

Since we're tracing in **:break** mode, the call to test signalled a continuable error. The error message shows the traced stack limit, the amount of stack memory currently in use, and offers you the option of resuming execution until the next increase in stack memory. To continue, simply type:

```
-> :c Continue until next stack increase.
#<FIELD-Pvar 9-8 *DEFAULT-VP-SET* (128 64)>
```

The pattern of tracing shown above is common enough that **trace-stack** includes the two shorthand options **:init** and **:reset** to reduce the number of function calls involved.

Calling trace-stack with the **:init** option calls ***warm–boot** and selects **:trace** mode:

```
(trace-stack :init)
Stack tracing is now on in :TRACE mode.
Current stack level is 1536.
Maximum stack limit is 1536.
1536
1536
```

And once we've run the function that we want to trace,

```
(test 9 3 2)
#<FIELD-Pvar 9-7 *DEFAULT-VP-SET* (128 64)>
```

we can call **trace–stack** with the :reset option to call ***warm–boot** again and then select **:break** tracing.

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

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

The output of the **:error** tracing mode is much the same as that of the **:break** tracing. The **:warn** mode's output, though, is a little different:

```
(trace-stack :warn)
Stack tracing is now on in :WARN mode.
Current stack level is 1554.
Maximum stack limit is 1554.
1554
1554
```

```
(!! 9)
;;; Warning:
;;; Stack has reached/exceeded traced maximum of 1554.
#<FIELD-Pvar 11-4 *DEFAULT-VP-SET* (128 64)
```

If you want, you can use the **:newmax** option at any time to make the current stack level be the new maximum stack limit for future tracing:

```
(trace-stack :newmax)
Stack tracing is now on in :WARN mode.
Current stack level is 1558.
Maximum stack limit is 1558.
1558
1558
```

If you prefer a warning only when stack usage exceeds the current limit, you can use the **:break–above–limit** option to switch to this style of tracing.

```
(trace-stack :break-above-limit)
Tracing will now signal ABOVE stack limit.
1558
1558

(!! 1)
;;; Warning:
;;; Stack has reached/exceeded traced maximum of 1558.
```

And you can use the **:break–at–limit** option to switch back:

```
(trace-stack :break-at-limit)
Tracing will now signal AT stack limit.
1559
1558
```

And when you're finished tracing, you'll want to turn the trace facility **:off**:

```
(trace-stack :off)
Stack tracing is now off.
Current stack level is 1559.
Maximum stack limit is 1558.
1559
1558
```

Notice that even though the trace facility is off, you can still use **trace–stack** to get a report of the current settings:

```
(*warm-boot)
(trace-stack :status)
Stack tracing is now off.
Current stack level is 1536.
Maximum stack limit is 1558.
1536
1558
```

You can also modify the value of the global variable ***maximum–stack–level*** to set the maximum stack limit "manually":

```
> *maximum-stack-level*
1558

(setq *maximum-stack-level* 1600)
(trace-stack :break)
Stack tracing is now on in :BREAK mode.
Current stack level is 1536.
Maximum stack limit is 1600.
1536
1600
```

This allows you to catch *Lisp programs that are "running away"—allocating large numbers of stack pvars and consequently running out of memory. For example:

```
(dotimes (i 100)
       (+!! 2 3 9))

>>Error:  Stack has reached/exceeded traced maximum of 1600.
          Stack is now at 1601.
*LISP-I::MAX-STACK-LEVEL-CHECK:
Original code:  (LUCID-COMMON-LISP:NAMED-LAMBDA...)
:C  0: Continue until next stack increase.
:A  1: Abort to Lisp Top Level
-> :a
Abort to Lisp Top Level
```

If you wish, you can even write your own stack-tracing function, and use the **:call** option to select it:

```
(defun trace-check (current max)
   (print (list current max)))
(setq *maximum-stack-function* 'trace-check)
```

This function simply prints out the current and maximum stack levels whenever a stack pvar is allocated. For example:

```
(*warm-boot)
(trace-stack :call)
Stack tracing is now on in :CALL mode.
Current stack level is 1536.
Maximum stack limit is 1600.
1536
1600
```

Now, when you type

```
(!! 9)
```

the following is displayed:

```
(1540 1600)
#<FIELD-Pvar 1-4 *DEFAULT-VP-SET* (128 64)>
```

NOTES

Usage Note:

If you select **:break**, **:error**, or **:warning** tracing without having previously done a **:trace** stack trace of your program, the current stack level is used as the value of the ***maximum-stack-level*** variable.

Performance Note:

Because the **trace–stack** operator works by wrapping an "advice" function around the Paris operator that allocates CM stack space, you will see a degradation of performance while stack tracing is active. When stack tracing is disabled, however, the "advice" function is removed, and performance returns to normal.

truncate!! [*Function*]

Performs a parallel truncation on the supplied pvar(s).

SYNTAX

truncate!! *numeric–pvar* **&optional** *divisor–numeric–pvar*

ARGUMENTS

numeric–pvar Non-complex numeric pvar. Pvar to be truncated.

divisor–numeric–pvar Non-complex numeric pvar. If supplied, *numeric–pvar* is divided by *divisor–numeric–pvar* before truncation is done.

RETURNED VALUE

truncate–pvar Temporary integer pvar. In each active processor, contains the truncated value of *numeric–pvar*, divided by *divisor–numeric–pvar* if supplied.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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

REFERENCES

See also these related rounding operations:

ceiling!! **floor!!** **round!!**

See also these related floating-point rounding operations:

fceiling!! **ffloor!!** **fround!!** **ftruncate!!**

typed–vector!! [*Function*]

Creates and returns a vector pvar of the specified type.

SYNTAX

typed–vector!! *component–type* **&rest** *component–pvars*

ARGUMENTS

component–type	*Lisp type specifier. Type of vector pvar to create.
component–pvars	Pvars containing values of type specified by *component–type*. Determine initial contents of returned vector.

RETURNED VALUE

typed–vector–pvar	Temporary vector pvar, of type specified by *component–type*. In each active processor, contains a vector whose elements are the corresponding values of the *component–pvar* arguments.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The function **typed–vector!!** creates and returns a one-dimensional array pvar of type *component–type*. The contents of the returned *typed–vector–pvar* are copied from the supplied *component–pvars*. In each processor, the *n*th element of the vector in *typed–vector–pvar* is a copy of the value of the *n*th *component–pvar* argument.

EXAMPLES

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

```
(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))))))
```

REFERENCES

See also the pvar allocation and deallocation operations

allocate!!	**array!!**	
***deallocate**	***deallocate–*defvars**	***defvar**
front–end!!	***let**	***let***
make–array!!	**vector!!**	**!!**

typep!! [*Function*]

Tests the values of a pvar in parallel for a specified scalar data type.

SYNTAX

typep!! *pvar scalar–type*

ARGUMENTS

pvar	Pvar expression. Pvar for which values are tested.
scalar–type	Type specifier. Data type for which values of *pvar* are tested.

RETURNED VALUE

typep–pvar	Temporary boolean pvar. Contains the value **t** in each active processor for which the value of *pvar* is of the data type specified by *scalar–type*. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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 returned *typep–pvar* pvar contains **t** in each processor where *pvar* is of type *scalar–type* and contains **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:** This is not supported by the *Lisp simulator for array pvars.

EXAMPLES

```
(typep!! (!! t) 'boolean) => t!!
```

These two invocations of **typep!!** both return **t** in processors 0 through 10 and **nil** elsewhere.

```
(typep!! (self-address!!) '(integer 0 10))
(typep!! (float!! (self-address!!)) '(float 0.0 10.0))
```

NOTES

No *Lisp equivalent of the Common Lisp **satisfies** type constructor is provided.

REFERENCES

See also these related pvar data type predicates:

booleanp!!	**characterp!!**	**complexp!!**
floatp!!	**front–end–p!!**	**integerp!!**
numberp!!	**string–char–p!!**	**structurep!!**

*undefsetf [*Function*]

Removes any update function bound to the specified parallel structure access function by ***defsetf.**

SYNTAX

***undefsetf** *accessor-function*

ARGUMENTS

accessor-function	Symbol. The name of an accessor function to a parallel structure, as created by ***defstruct.**

RETURNED VALUE

nil	Evaluated for side effect only.

SIDE EFFECTS

Removes any update function bound to *accessor-function* by ***defsetf.**

DESCRIPTION

This function removes from the supplied *accessor-function* any update-function bindings created by ***defsetf.**

REFERENCES

See also these related operations:

***defsetf** ***setf**

un*defun [*Function*]

Undefines functions defined using ***defun**.

SYNTAX

un*defun &rest **defun–names*

ARGUMENTS

**defun–names*	Symbols. Names of functions defined with ***defun** to undefine.

RETURNED VALUE

nil	Evaluated for side effect only.

SIDE EFFECTS

Removes all macros and functions bound to the supplied symbol names by ***defun**.

DESCRIPTION

Removes the macro binding from each specified ***defun** name and removes the function binding from all symbols derived from the ***defun** names.

The **&rest** arguments must be the symbolic names of functions that have previously been defined with ***defun**. Any number of names may be provided.

When **(*defun foo ...)** is called, both a macro named **foo** and a function with a name derived from **foo** are created. A call to **(un*defun foo)** undefines both the macro and the associated function.

REFERENCES

See also the following related operations:

***apply**	***defun**	***funcall**
***trace**	***untrace**	

*unless [*Macro*]

Evaluates *Lisp forms with the currently selected set bound according to the logical value of a pvar expression.

SYNTAX

***unless** *test–pvar* **&body** *body*

ARGUMENTS

test–pvar	Pvar expression. Selects processors in which to evaluate body.
body	*Lisp forms. Evaluated with the currently selected set restricted to those processors in which the value of *test–pvar* is **nil**.

RETURNED VALUE

body–value	Scalar or pvar value. Value of final form in *body.*

SIDE EFFECTS

Temporarily restricts the currently selected set during the evaluation of the forms in *body.*

DESCRIPTION

The ***unless** macro evaluates the supplied *body* forms with the currently selected set bound so that only processors in which *test–pvar* is **nil** are selected. The ***unless** macro subselects from the currently selected set of processors, so that any processor that is unselected when ***unless** is called remains unselected during the evaluation of the *body* forms. All forms in the *body* are evaluated, even if no processors are selected. The value of the final expression in the *body* is returned whether it is a Lisp value or a pvar.

EXAMPLES

The ***unless** form is similar to a call to the ***when** form with the *test–pvar* is negated. Thus:

```
(*unless unworthy-pvar . . .)
<=>
(*when (not!! unworthy-pvar) . . .)
```

This example increments the value of **price–of–movie–pvar** in all processors where **age–pvar** is greater than or equal to 12.

```
(*unless (<!! age-pvar (!! 12))
 (*incf price-of-movie-pvar (!! 3))
```

NOTES

Usage Note:

Forms such as **throw**, **return**, **return–from**, and **go** may be used to exit an external block or looping construct from within a processor selection operator. However, doing so will leave the currently selected set in the state it was in at the time the non-local exit form is executed. To avoid this, use the *Lisp macro **with–css–saved**. For example,

```
(block division
   (with-css-saved
      (*unless (<=!! y (!! 0))
        (if (*or (=!! (!! 0) x))
            (return-from division nil)
            (/!! y x)))))
```

Here **return–from** is used to exit from the **division** block if the value of **x** in any processor is zero. When the **with–css–saved** macro is entered, it saves the state of the currently selected set. When the code enclosed within the **with–css–saved** exits for any reason, either normally or via a call to an non-local exit operator like **return–from**, the currently selected set is restored to its original state.

See the dictionary entry for **with–css–saved** for more information.

REFERENCES

See also the related operators

***all**	***case**	**case!!**	***cond**	**cond!!**	
***ecase**	**ecase!!**	***if**	**if!!**	***when**	**with–css–saved**

unproclaim [*Function*]

Removes a global declaration previously made with ***proclaim**.

SYNTAX

unproclaim *declaration*

ARGUMENTS

declaration	*Lisp declaration form previously supplied as argument to ***proclaim**. Global declaration to be removed.

RETURNED VALUE

nil	Evaluated for side effect only.

SIDE EFFECTS

Removes the global declaration specified by *declaration*.

DESCRIPTION

Removes the effects of a declaration made with ***proclaim**.

REFERENCES

See also the related *Lisp declaration operators:

***locally** ***proclaim**

See also the related type translation function **taken-as!!**.

See also the related type coercion function **coerce!!**.

*untrace [*Macro*]

Cancels tracing for the specified user-defined *Lisp functions.

SYNTAX

***untrace &rest** **defun–function–names*

ARGUMENTS

**defun–function–names*
Symbols. Names of user-defined *Lisp functions for which tracing is to be cancelled.

RETURNED VALUE

traced–functions List of symbols. Names of functions untraced.

SIDE EFFECTS

Cancels tracing on the named functions. Has no effect on functions which are not currently traced.

DESCRIPTION

Cancels tracing for the named parallel functions, which must have been defined using ***defun.**

EXAMPLES

Invoked at top level, **(*trace foo)** causes a message to be printed whenever the function **foo** is either called or exited. For example,

```
(*defun self-random!! ()
  (random!! (1+!! (self-address!!))))

(*trace self-random!!)  =>  (*DEFUN-SELF-RANDOM!!)

(self-random!!)  =>
1 Enter *DEFUN-SELF-RANDOM!!
1 Exit *DEFUN-SELF-RANDOM!! #<Structure PVAR A032B6>
#<Structure PVAR A03276>
```

A call **(*untrace self–random!!)** turns off this tracing mechanism.

```
(*untrace self-random!!)  =>  (*DEFUN-SELF-RANDOM!!)
```

REFERENCES

The macros ***trace** and ***untrace** are the parallel equivalents of the Common Lisp **trace** and **untrace** functions, defined in *Common Lisp: The Language.*

See also the following related operations:

***apply**	***defun**	***funcall**
un*defun		

upper–case–p!! [*Function*]

Performs a parallel test for uppercase characters on the supplied pvar.

SYNTAX

upper–case–p!! *character–pvar*

ARGUMENTS

character–pvar Character pvar. Tested in parallel for uppercase characters.

RETURNED VALUE

uppercasep–pvar Temporary boolean pvar. Contains the value **t** in each active processor where the corresponding value of *character–pvar* is an uppercase alphabetic character. Contains **nil** in all other processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This predicate returns a pvar that has the value **t** in each processor where the supplied *character–pvar* contains an uppercase character, and the value **nil** in all other processors.

v+, v–, v*, v/ [*Function*]

Return the vector sum, difference, product, or quotient of the supplied front-end vectors.

SYNTAX

v+, v–, v*, v/ *vector* **&rest** *vectors*

ARGUMENTS

vector, vectors	Front-end vectors. All vectors supplied must have the same element size.

RETURNED VALUE

result–vector	Front-end vector. The combination of the supplied arguments.

SIDE EFFECTS

None.

DESCRIPTION

These operations are the serial (front end) equivalents of **v+!!, v–!!, v*!!, v/!!**.

REFERENCES

This function is one of a number of front-end vector operators, listed below:

cross–product	**dot–product**	**v+ v– v* v/**	
v+–constant	**v—constant**	**v*–constant**	**v/–constant**
vabs	**vabs–squared**	**vceiling**	**vector–normal**
vfloor	**vround**	**vscale**	
vscale–to–unit–vector		**vtruncate**	

These functions are the serial equivalents of the corresponding vector pvar operations. See Chapter 1, "*Lisp Overview," of this Dictionary for a list of these functions.

---

v+!!, v–!!, v*!!, v/!! [*Function*]

Calculate in parallel the vector sum, difference, product, or quotient of vector pvars.

SYNTAX

v+!!, v–!!, v*!!, v/!! *vector–pvar* **&rest** *vector–pvars*

ARGUMENTS

vector–pvar, vector–pvars
Vector pvars. Pvars for which vector combination is calculated. All pvars supplied must have the same element size.

RETURNED VALUE

result–vector–pvar Temporary vector pvar. In each active processor, contains the result of combining *vector–pvar* with the corresponding values of the *vector–pvars*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The **v+!!, v–!!, v*!!,** and **v/!!** functions calculate in each processor the element-wise vector combination of the values of the supplied *vector–pvars*. If only a single argument is supplied, its values are simply copied into the returned *result–pvar.*

EXAMPLES

The following equivalences hold:

```
(v+!! vector-pvar-1 vector-pvar-2 ... vector-pvar-n)
    <=>
(amap!! '+!! vector-pvar-1 vector-pvar-2 ... vector-pvar-n)
```

```
(v-!! vector-pvar-1 vector-pvar-2 ... vector-pvar-n)
    <=>
(amap!! '-!! vector-pvar-1 vector-pvar-2 ... vector-pvar-n)


(v*!! vector-pvar-1 vector-pvar-2 ... vector-pvar-n)
    <=>
(amap!! '*!! vector-pvar-1 vector-pvar-2 ... vector-pvar-n)


(v/!! vector-pvar-1 vector-pvar-2 ... vector-pvar-n)
    <=>
(amap!! '/!! vector-pvar-1 vector-pvar-2 ... vector-pvar-n)
```

REFERENCES

This function is one of a number of related vector pvar operators, listed below:

cross–product!!	**dot–product!!**	**v+!! v–!! v*!! v/!!**	
v+scalar!!	**v–scalar!!**	**v*scalar!!**	**v/scalar!!**
vabs!!	**vabs–squared!!**	**vector–normal!!**	**vscale!!**
vscale–to–unit–vector!!		***vset–components**	

v{+,–,*,/}–constant [*Function*]

Combine a scalar value with each element of a vector.

SYNTAX

v+–constant **v—constant** **v*–constant** **v/–constant** *vector scalar*

ARGUMENTS

vector	Front-end vector. Vector with which *scalar* is combined.
scalar	Scalar value. Combined with each vector element of *vector.*

RETURNED VALUE

result–vector	Scalar vector. Result of combining *scalar* with elements of *vector*.

SIDE EFFECTS

None.

DESCRIPTION

These are the serial equivalents of **v+scalar!!**, **v–scalar!!**, **v*scalar!!**, and **v/scalar!!**.

REFERENCES

This function is one of a number of front-end vector operators, listed below:

cross–product	**dot–product**	**v+ v– v* v/**	
v+–constant	**v—constant**	**v*–constant**	**v/–constant**
vabs	**vabs–squared**	**vceiling**	**vector–normal**
vfloor **vround**	**vscale**	**vscale–to–unit–vector**	**vtruncate**

These functions are the serial equivalents of the corresponding vector pvar operations. See Chapter 1, "*Lisp Overview," of this Dictionary for a list of these functions.

---

v+scalar!!, v–scalar!!, v*scalar!!, v/scalar!!

[*Function*]

Perform an elementwise arithmetic operation on a vector pvar.

SYNTAX

v+scalar!! **v–scalar!!** **v*scalar!!** **v/scalar!!** *vector–pvar scalar–pvar*

ARGUMENTS

vector–pvar Vector pvar. Pvar on which elementwise operation is performed.

scalar–pvar Non-aggregate pvar. Value by which each element of *vector–pvar* is modified.

RETURNED VALUE

vector–pvar Temporary vector pvar. Copy of *vector–pvar* in which each element has been modified by the value of *scalar–pvar.*

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

In each processor, these functions perform an elementwise arithmetic operation on the vector in *vector–pvar*, as follows:

- **v+scalar!!** adds the value of scalar-pvar to each element of vector-pvar.
- **v–scalar!!** subtracts the value of scalar-pvar from each element of vector-pvar.
- **v*scalar!!** multiplies each element of vector-pvar by the value of scalar-pvar.
- **v/scalar!!** divides each element of vector-pvar by the value of scalar-pvar.

EXAMPLES

```
(v+scalar!! (!! #(1 2 3)) (!! 3))  <=>  (!! #(4 5 6))
(v-scalar!! (!! #(4 5 6)) (!! 3))  <=>  (!! #(1 2 3))
(v*scalar!! (!! #(1 2 3)) (!! 3))  <=>  (!! #(3 6 9))
(v/scalar!! (!! #(3 6 9)) (!! 3))  <=>  (!! #(1.0 2.0 3.0))
```

NOTES

These functions are generalized versions of the now obsolete single-float vector pvar operations **sf–v+–constant!!**, **sf–v—constant!!**, **sf–v*–constant!!**, and **sf–v/–constant!!**. The term "scalar" is used rather than "constant" for accuracy, as the scalar-pvar argument to any one of these operations is not constrained to contain a constant value in all processors.

REFERENCES

This function is one of a number of related vector pvar operators, listed below:

cross–product!!	**dot–product!!**	**v+!! v–!! v*!! v/!!**	
v+scalar!!	**v–scalar!!**	**v*scalar!!**	**v/scalar!!**
vabs!!	**vabs–squared!!**	**vector–normal!!**	**vscale!!**
vscale–to–unit–vector!!		***vset–components**	

vabs [*Function*]

Returns the vector magnitude of the supplied front-end vector.

SYNTAX

vabs *vector*

ARGUMENTS

vector	Front-end vector. Vector for which magnitude is returned.

RETURNED VALUE

vector–length	Numeric value. Magnitude of *vector*.

SIDE EFFECTS

None.

DESCRIPTION

This is the serial (front end) equivalent of **vabs!!**. This function is equivalent to

```
(sqrt (vabs-squared vector))
```

REFERENCES

This function is one of a number of front-end vector operators, listed below:

cross–product	**dot–product**	**v+ v– v* v/**	
v+–constant	**v—constant**	**v*–constant**	**v/–constant**
vabs	**vabs–squared**	**vceiling**	**vector–normal**
vfloor vround	**vscale**	**vscale–to–unit–vector**	**vtruncate**

These functions are the serial equivalents of the corresponding vector pvar operations. See Chapter 1, "*Lisp Overview," of this Dictionary for a list of these functions.

vabs!! *[Function]*

Calculates in parallel the vector magnitude of the supplied vector pvar.

SYNTAX

vabs!! *vector–pvar*

ARGUMENTS

vector–pvar	Vector pvar. Pvar for which vector magnitude is computed.

RETURNED VALUE

result–pvar	Temporary vector pvar. In each active processor, contains the vector magnitude of the corresponding value of *vector–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function is equivalent to

```
(sqrt!! (vabs-squared!! vector-pvar))
```

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.

NOTES

Compiler Note:

The *Lisp compiler does not compile this operation.

REFERENCES

This function is one of a number of related vector pvar operators, listed below:

cross–product!!	**dot–product!!**	**v+!! v–!! v*!! v/!!**	
v+scalar!!	**v–scalar!!**	**v*scalar!!**	**v/scalar!!**
vabs!!	**vabs–squared!!**	**vector–normal!!**	**vscale!!**
vscale–to–unit–vector!!		***vset–components**	

vabs–squared [*Function*]

Returns the squared magnitude of the supplied front-end vector.

SYNTAX

vabs–squared *vector*

ARGUMENTS

vector	Front-end vector. Vector for which squared magnitude is returned.

RETURNED VALUE

vector–square	Numeric value. Squared magnitude of *vector*.

SIDE EFFECTS

None.

DESCRIPTION

This is the serial (front end) equivalent **vabs–squared!!**. This function is equivalent to the expression **(dot–product vector vector)**.

REFERENCES

This function is one of a number of front-end vector operators, listed below:

cross–product	**dot–product**	**v+ v– v* v/**	
v+–constant	**v—constant**	**v*–constant**	**v/–constant**
vabs	**vabs–squared**	**vceiling**	**vector–normal**
vfloor vround	**vscale**	**vscale–to–unit–vector**	**vtruncate**

These functions are the serial equivalents of the corresponding vector pvar operations. See Chapter 1, "*Lisp Overview," of this Dictionary for a list of these functions.

---

vabs–squared!! [*Function*]

Calculates in parallel the squared magnitude of the supplied vector pvar.

SYNTAX

vabs–squared!! *vector–pvar*

ARGUMENTS

vector–pvar Vector pvar. Pvar for which squared magnitude is computed.

RETURNED VALUE

result–pvar Temporary vector pvar. In each active processor, contains the squared magnitude of the corresponding value of *vector–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The **vabs–squared!!** function calculates in parallel the squared magnitude of the supplied *vector–pvar*. The *result–pvar* is of the same type as the supplied *vector–pvar*, but may be of larger size if *vector–pvar* is an unsigned or signed integer pvar.

Calling **(vabs–squared!!** *vector–pvar***)** is equivalent to

```
(dot-product!! vector-pvar vector-pvar)
```

NOTES

Compiler Note:

The *Lisp compiler does not compile this operation.

REFERENCES

This function is one of a number of related vector pvar operators, listed below:

cross–product!!	**dot–product!!**	**v+!! v–!! v*!! v/!!**	
v+scalar!!	**v–scalar!!**	**v*scalar!!**	**v/scalar!!**
vabs!!	**vabs–squared!!**	**vector–normal!!**	**vscale!!**
vscale–to–unit–vector!!		***vset–components**	

vceiling [*Function*]

Takes the ceiling of the supplied front-end vector.

SYNTAX

vceiling *vector*

ARGUMENTS

vector Front-end vector. Vector for which ceiling is taken.

RETURNED VALUE

vector–ceiling Vector. Elementwise ceiling of *vector*.

SIDE EFFECTS

None.

DESCRIPTION

Takes the ceiling of each element of *vector*.

REFERENCES

This function is one of a number of front-end vector operators, listed below:

cross–product	**dot–product**	**v+ v– v* v/**	
v+–constant	**v—constant**	**v*–constant**	**v/–constant**
vabs	**vabs–squared**	**vceiling**	**vector–normal**
vfloor vround	**vscale**	**vscale–to–unit–vector**	**vtruncate**

These functions are the serial equivalents of the corresponding vector pvar operations. See Chapter 1, "*Lisp Overview," of this Dictionary for a list of these functions.

vector!! [*Function*]

Creates and returns a vector pvar containing the values of the supplied pvars.

SYNTAX

vector!! &rest *element–pvars*

ARGUMENTS

element–pvars	Pvars. Used to initialize the returned vector pvar.

RETURNED VALUE

vector–pvar	Vector pvar. In each active processor, contains a vector whose elements are the corresponding values of the *element–pvars.*

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

Creates and returns a vector pvar, initialized with the values of the supplied *element–pvars.*

The standard rules of coercion are used to determine the element type of the resulting vector pvar. For instance, a mixture of integer and floating point elements yields a floating-point result. A mixture of floating-point and complex elements yields a complex result. An error is signaled if the data types present are not all compatible. (For instance, a string-char element and a floating-point element are not compatible.)

EXAMPLES

```
(pref (vector!! (self-address!!) (self-address!!)) 25)
=> #(25 25).
```

REFERENCES

The **vector!!** function is similar to the **typed–vector!!** function. However an element-type argument is not required for **vector!!**. See the definition of **typed–vector!!** for more information.

See also the pvar allocation and deallocation operations

allocate!!	**array!!**	
***deallocate**	***deallocate–*defvars**	***defvar**
front–end!!	***let**	***let***
make–array!!	**!!**	

vector–normal [*Function*]

Returns the normalized cross product of two front-end vectors.

SYNTAX

vector–normal *vector1 vector2*

ARGUMENTS

vector1, vector2 Front-end vectors. Vectors for which normalized cross product is calculated. Vectors must be at least 3 elements in length.

RETURNED VALUE

normal–vector Front-end vector. Normalized cross product of *vector1* and *vector2*.

SIDE EFFECTS

None.

DESCRIPTION

This is the serial (front end) equivalent of **vector–normal!!**.

This function is equivalent to

```
(vscale-to-unit-vector
   (cross-product vector-pvar1 vector-pvar2))
```

REFERENCES

This function is one of a number of front-end vector operators, listed below:

cross–product	**dot–product**	**v+ v– v* v/**	
v+–constant	**v—constant**	**v*–constant**	**v/–constant**
vabs	**vabs–squared**	**vceiling**	**vector–normal**
vfloor vround	**vscale**	**vscale–to–unit–vector**	**vtruncate**

These functions are the serial equivalents of the corresponding vector pvar operations. See Chapter 1, "*Lisp Overview," of this Dictionary for a list of these functions.

vector–normal!! [*Function*]

Calculates in parallel the normalized cross-product of the supplied vector pvars.

SYNTAX

vector–normal!! *vector–pvar–1 vector–pvar–2*

ARGUMENTS

vector–pvar–1, vector–pvar–2
: Vector pvars. Pvars for which normalized cross-product is calculated.

RETURNED VALUE

vector–normal–pvar
: Temporary vector pvar. In each active processor, contains the normalized cross-product of the corresponding values of *vector–pvar1* and *vector–pvar2*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function calculates in parallel the normalized cross-product of two single-float vector pvars, and is equivalent to

```
(vscale-to-unit-vector!!
   (cross-product!! vector-pvar1 vector-pvar2))
```

EXAMPLES

```
(vector-normal!!
    (!! #(1 0 0)) (!! #(0 1 0))  <=>  (!! #(0.0 0.0 1.0))

(vector-normal!!
    (!! #(0 1 0)) (!! #(1 0 0))  <=>  (!! #(0.0 0.0 -1.0))
```

NOTES

Usage note:

The orientation of the normalized cross#product produced in each processor depends on the order of the vector-pvar arguments. Specifically,

```
(*set v1 (vector-normal!! vector-pvar1 vector-pvar2))
(*set v2 (vector-normal!! vector-pvar2 vector-pvar1))

v1  <=>  (v*scalar!! v2 (!! -1))
```

that is, v1 is the vector negative of v2.

REFERENCES

This function is one of a number of related vector pvar operators, listed below:

cross–product!!	**dot–product!!**	**v+!! v–!! v*!! v/!!**	
v+scalar!!	**v–scalar!!**	**v*scalar!!**	**v/scalar!!**
vabs!!	**vabs–squared!!**	**vector–normal!!**	**vscale!!**
vscale–to–unit–vector!!		***vset–components**	

vfloor [*Function*]

Takes the floor of the supplied front-end vector.

SYNTAX

vfloor *vector*

ARGUMENTS

vector	Front-end vector. Vector for which floor is taken.

RETURNED VALUE

vector–floor	Vector. Elementwise floor of *vector*.

SIDE EFFECTS

None.

DESCRIPTION

Takes the floor of each element of *vector*.

REFERENCES

This function is one of a number of front-end vector operators, listed below:

cross–product	**dot–product**	**v+ v– v* v/**	
v+–constant	**v—constant**	**v*–constant**	**v/–constant**
vabs	**vabs–squared**	**vceiling**	**vector–normal**
vfloor vround	**vscale**	**vscale–to–unit–vector**	**vtruncate**

These functions are the serial equivalents of the corresponding vector pvar operations. See Chapter 1, "*Lisp Overview," of this Dictionary for a list of these functions.

---

vp–set–deallocated–p, vp–set–dimensions vp–set–rank, vp–set–total–size, vp–set–vp–ratio [*Function*]

Return information about the specified *vp–set.*

SYNTAX

vp–set–deallocated–p *vp–set*
vp–set–dimensions *vp–set*
vp–set–rank *vp–set*
vp–set–total–size *vp–set*
vp–set–vp–ratio *vp–set*

ARGUMENTS

vp–set	VP set object.

RETURNED VALUE

Each of these functions returns a single value, as described below:

deallocated–p	Boolean. The value **t** if *vp–set* is deallocated, and **nil** otherwise.
dimensions–list	List of integers. Dimensions of the supplied VP set.
rank	Integer. The rank, or number of dimensions, of the supplied VP set.
total–size	Integer. Total number of processors in *vp–set.*
vp–ratio	Integer. The VP ratio (number of virtual processors per physical processor) of the VP set.

SIDE EFFECTS

None.

DESCRIPTION

Each of these functions returns information about the supplied *vp–set* argument, as described in the Returned Value section above.

NOTES

The *vp–set* argument must be a *Lisp VP set, created by a call to a *Lisp operator such as **def–vp–set** or **create–vp–set**.

REFERENCES

See also the following VP set information operations:

dimension–size **dimension–address–length**
describe–vp–set

vround [Function]

Rounds the supplied front-end vector.

SYNTAX

vround *vector*

ARGUMENTS

vector	Non-complex numeric vector. Vector for which round is taken.

RETURNED VALUE

vector–round	Numeric value. Rounded value of *vector*.

SIDE EFFECTS

None.

DESCRIPTION

This function rounds each element of *vector* to the nearest integer.

REFERENCES

This function is one of a number of front-end vector operators, listed below:

cross–product	**dot–product**	**v+ v– v* v/**	
v+–constant	**v—constant**	**v*–constant**	**v/–constant**
vabs	**vabs–squared**	**vceiling**	**vector–normal**
vfloor vround	**vscale**	**vscale–to–unit–vector**	**vtruncate**

These functions are the serial equivalents of the corresponding vector pvar operations. See Chapter 1, "*Lisp Overview," of this Dictionary for a list of these functions.

vscale

[*Function*]

Scales a front-end vector by a scalar value.

SYNTAX

vscale *vector scalar*

ARGUMENTS

vector	Front-end vector. Vector to be scaled.
scalar	Scalar value. Value by which to scale *vector*.

RETURNED VALUE

scaled–vector	The result of multiplying each element of *vector* by *scalar*.

DESCRIPTION

This is the serial (front end) equivalent of **vscale!!**.

REFERENCES

This function is one of a number of front-end vector operators, listed below:

cross–product	**dot–product**	**v+ v– v* v/**	
v+–constant	**v—constant**	**v*–constant**	**v/–constant**
vabs	**vabs–squared**	**vceiling**	**vector–normal**
vfloor vround	**vscale**	**vscale–to–unit–vector**	**vtruncate**

These functions are the serial equivalents of the corresponding vector pvar operations. See Chapter 1, "*Lisp Overview," of this Dictionary for a list of these functions.

---

vscale!! [*Function*]

Calculates in parallel the result of scaling the supplied vector pvar by a scalar pvar.

SYNTAX

vscale!! *vector–pvar scalar–pvar*

ARGUMENTS

vector–pvar	Vector pvar. Vector pvar to be scaled.
scalar–pvar	Scalar pvar. Value by which *vector–pvar* is scaled.

RETURNED VALUE

result–pvar	Temporary vector pvar. In each active processor, contains the result of scaling each element of the vector in *vector–pvar* by the value in *scalar–pvar*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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.

REFERENCES

This function is one of a number of related vector pvar operators, listed below:

cross–product!!	**dot–product!!**	**v+!! v–!! v*!! v/!!**	
v+scalar!!	**v–scalar!!**	**v*scalar!!**	**v/scalar!!**
vabs!!	**vabs–squared!!**	**vector–normal!!**	**vscale!!**
vscale–to–unit–vector!!		***vset–components**	

vscale–to–unit–vector [*Function*]

Returns the result of scaling the supplied front-end vector to unit length.

SYNTAX

vscale–to–unit–vector *vector*

ARGUMENTS

vector	Front-end vector. Vector to be scaled.

RETURNED VALUE

unit–vector	Front-end vector. The result of scaling *vector* to a unit-length vector.

SIDE EFFECTS

None.

DESCRIPTION

This is the serial (front end) equivalent of **vscale–to–unit–vector!!**.

This function is equivalent to

```
(vscale vector (/ (vabs vector)))
```

except that *vector* is evaluated once.

NOTES

It is an error if every element of *vector* is 0, or if **(vabs** *vector* **)** is 0.

REFERENCES

This function is one of a number of front-end vector operators, listed below:

cross–product	**dot–product**	**v+ v– v* v/**	
v+–constant	**v—constant**	**v*–constant**	**v/–constant**
vabs	**vabs–squared**	**vceiling**	**vector–normal**
vfloor vround	**vscale**	**vscale–to–unit–vector**	**vtruncate**

These functions are the serial equivalents of the corresponding vector pvar operations. See Chapter 1, "*Lisp Overview," of this Dictionary for a list of these functions.

vscale–to–unit–vector!! *[Function]*

Calculates in parallel the result of scaling the supplied vector pvar to unit length.

SYNTAX

vscale–to–unit–vector!! *vector–pvar*

ARGUMENTS

vector–pvar	Vector pvar. Vector pvar to be scaled.

RETURNED VALUE

result–pvar	Temporary vector pvar. In each active processor, contains the result of scaling the value of *vector–pvar* to a unit-length vector.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This function is equivalent to

```
(vscale!! vector-pvar (/!! (vabs!! vector-pvar)))
```

except that *vector–pvar* is evaluated once.

EXAMPLES

It is an error if *vector–pvar* contains a zero-length vector in any active processor.

NOTES

Compiler Note:

The *Lisp compiler does not compile this operation.

REFERENCES

This function is one of a number of related vector pvar operators, listed below:

cross–product!!	**dot–product!!**	**v+!! v–!! v*!! v/!!**	
v+scalar!!	**v–scalar!!**	**v*scalar!!**	**v/scalar!!**
vabs!!	**vabs–squared!!**	**vector–normal!!**	**vscale!!**
vscale–to–unit–vector!!		***vset–components**	

*vset–components [*Defun]

Copies the supplied element pvars into the supplied vector pvar in parallel.

SYNTAX

***vset–components** *vector–pvar* **&rest** *element–pvars*

ARGUMENTS

vector–pvar	Vector pvar. Vector pvar into which elements are stored.
element–pvars	Pvar(s) of same type as elements of *vector–pvar*. Element(s) to be stored. Either a single pvar or as many pvars as there are elements in *vector–pvar*.

RETURNED VALUE

nil	Evaluated for side effect only.

SIDE EFFECTS

Destructively alters the value of *vector–pvar* in each active processor to contain the elements specified by the supplied *element–pvars*.

DESCRIPTION

This function copies the values of the supplied *element–pvars* into the supplied *vector–pvar* in parallel.

If there is a single *element–pvar* argument, then every element of *vector–pvar* is ***set** to it. If there are as many *element–pvar* arguments as there are elements in *vector–pvar*, then the jth element of *vector–pvar* is ***set** to the jth *element–pvar* argument. An error will be signaled if the number of *element–pvar* arguments is not either 1 or the number of elements in the *vector–pvar*.

REFERENCES

This function is one of a number of related vector pvar operators, listed below:

cross–product!!	**dot–product!!**	**v+!! v–!! v*!! v/!!**	
v+scalar!!	**v–scalar!!**	**v*scalar!!**	**v/scalar!!**
vabs!!	**vabs–squared!!**	**vector–normal!!**	**vscale!!**
vscale–to–unit–vector!!		***vset–components**	

vtruncate [Function]

Truncates the supplied front-end vector.

SYNTAX

vtruncate *vector*

ARGUMENTS

vector	Front-end vector. Vector to be truncated.

RETURNED VALUE

vector–truncate	Numeric value. Truncated value of *vector*.

SIDE EFFECTS

None.

DESCRIPTION

Truncates each element of *vector*.

REFERENCES

This function is one of a number of front-end vector operators, listed below:

cross–product	**dot–product**	**v+ v– v* v/**	
v+–constant	**v—constant**	**v*–constant**	**v/–constant**
vabs	**vabs–squared**	**vceiling**	**vector–normal**
vfloor vround	**vscale**	**vscale–to–unit–vector**	**vtruncate**

These functions are the serial equivalents of the corresponding vector pvar operations. See Chapter 1, "*Lisp Overview," of this Dictionary for a list of these functions.

---

*warm–boot [*Macro*]

Clears the *Lisp stack, deallocating local and temporary pvars, resets the currently selected set of all VP sets, and resets certain internal states of the CM.

SYNTAX

***warm–boot**

ARGUMENTS

Takes no arguments.

RETURNED VALUE

nil	Evaluated for side effects only.

SIDE EFFECTS

Clears the *Lisp stack, deallocating local and temporary pvars, resets the currently selected set of all VP sets, and resets certain internal states of the CM.

DESCRIPTION

The ***warm–boot** macro resets *Lisp and the CM. It must be called whenever the CM has been placed in an inconsistent state, such as by a program error or by manually aborting a running function.

The ***warm–boot** macro clears the *Lisp stack and restores both *Lisp and the CM to a consistent, usable state. The *Lisp heap is not cleared, so pvars allocated on the heap (permanent and global) remain allocated.

Specifically, executing ***warm–boot** has the following effects:

- All virtual processors in all VP sets are made active; no processors remain unselected.
- The ***default–vp–set*** is selected as the ***current–vp–set***

- The Connection Machine stack is cleared and all pvars allocated on the stack (i.e., any not created by **allocate!!** or ***defvar**) are deallocated.

EXAMPLES

A top-level call to ***warm-boot** resets the CM.

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

The following example demonstrates why it is necessary to call ***warm-boot** after aborting execution in the middle of a *Lisp program.

```
(*cold-boot :initial-dimensions '(512))

(*let (x)
    (declare (type single-float-pvar x))
    (*when (evenp!! (self-address!!)) (*set x (!! #\x))))

Error: In interpreted *SET.
The source expression in a float-general *set contains something
that is not a float.
A pvar of type STRING-CHAR caused the error.

-> (*sum (!! 1))
256
```

The error occurs while the currently selected set has been restricted to the even processors. The following example shows that the currently selected set is not automatically restored by aborting back to top level.

```
-> Abort
Return to Lisp Top Level in Dynamic Lisp Listener 1
Back to Lisp Top Level in Dynamic Lisp Listener 1.

(*sum (!! 1))  =>  256
```

A call to ***warm-boot** resets the currently selected set so that all processors are active.

```
(*warm-boot)
(*sum (!! 1))  =>  512
```

Interrupting or aborting a program may leave the front-end/CM connection in an inconsistent state, preventing the front end from issuing instructions to the CM. A call to ***warm-boot** resets the connection, allowing the front end to communicate with the CM.

NOTES

The ***warm–boot** macro is intended to be called at top level. It should not in general be called from within user code, because it forcibly deallocates any existing local and temporary pvars.

One exception to this rule is that ***warm–boot** may be called as the first body form of a function intended to be called at top level, as in

```
(defun top-level ()
   (*warm-boot)
   (initialize-pvars)
   (main-function)
   (clean-up-and-print-results)
  )
```

Here, ***warm–boot** is used to ensure that the Connection Machine is reset and ready for use before the initialization function and main functions of the user's program are called.

REFERENCES

See also the related Connection Machine initialization operator ***cold–boot**.

See also the initialization-list functions **add–initialization** and **delete–initialization**.

See also the character attribute initialization operator **initialize–character**.

*when [*Macro*]

Evaluates *Lisp forms with the currently selected set bound according to the logical value of a pvar expression.

SYNTAX

***when** *test–pvar* **&body** *body*

ARGUMENTS

test–pvar	Pvar expression. Selects processors in which to evaluate body.
body	*Lisp forms. Evaluated with the currently selected set restricted to those processors in which the value of *test–pvar* is **t**.

RETURNED VALUE

body–value	Scalar or pvar value. Value of final form in *body.*

SIDE EFFECTS

Temporarily restricts the currently selected set during the evaluation of the forms in *body.*

DESCRIPTION

The ***when** macro evaluates the supplied *body* forms with the currently selected set bound so that only processors in which *test–pvar* is non-**nil** are selected. The ***when** macro subselects from the currently selected set of processors, so that any processor that is unselected when ***when** is called remains unselected during the evaluation of the *body* forms. All forms in the *body* are evaluated, even if no processors are selected. The value of the final expression in the *body* is returned, whether it is a Lisp value or a pvar.

EXAMPLES

This example increments the value of **price-of-movie-pvar** in all processors where **age-pvar** is greater than or equal to 12.

```
(*when (>=!! age-pvar (!! 12))
 (*incf price-of-movie-pvar (!! 3))
```

This example shows how ***when** may be nested to select processors in which a data pvar meets multiple criteria. The vaue of **intensity-pvar** is copied into **real-edge-pvar** only in those processors where **part-of-edge-p** is non-**nil**, and where **intensity-pvar** is greater than 9.0.

```
(*when part-of-edge-p
   (*when (>!! intensity-pvar (!! 9.0))
      (*set real-edge-pvar intensity-pvar)))
```

NOTES

Usage Note:

Forms such as **throw, return, return-from,** and **go** may be used to exit an external block or looping construct from within a processor selection operator. However, doing so will leave the currently selected set in the state it was in at the time the non-local exit form is executed. To avoid this, use the *Lisp macro **with-css-saved**. For example,

```
(block division
   (with-css-saved
      (*when (>!! y (!! 0))
        (if (*or (=!! (!! 0) x))
            (return-from division nil)
            (/!! y x)))))
```

Here **return-from** is used to exit from the **division** block if the value of **x** in any processor is zero. When the **with-css-saved** macro is entered, it saves the state of the currently selected set. When the code enclosed within the **with-css-saved** exits for any reason, either normally or via a call to a non-local exit operator like **return-from**, the currently selected set is restored to its original state.

See the dictionary entry for **with-css-saved** for more information.

Implementation Note:

If the last *body* form is either a ***all** or a ***when** form, then the inner form does not save/restore the state of the current selected set. This is mainly an optimization feature—it does not change the semantics of your code.

REFERENCES

See also the related operators

***all**	
***case**	**case!!**
***cond**	**cond!!**
***ecase**	**ecase!!**
***if**	**if!!**
***unless**	**with–css–saved**

with–css–saved [*Macro*]

Records the state of the currently selected set and ensures that it is automatically restored when evaluation of the supplied body forms terminates.

SYNTAX

with–css–saved &body *body*

ARGUMENTS

body	*Lisp forms. Body forms to be evaluated.

RETURNED VALUE

body–value	Scalar or pvar value. Value returned by final form in *body.*

SIDE EFFECTS

Records the state of the currently selected set before evaluating the forms in *body* and ensures that the currently selected set is restored when evaluation of the *body* forms terminates.

DESCRIPTION

The **with–css–saved** macro records the state of the currently selected set and ensures that when evaluation of the supplied *body* forms terminates for any reason, the recorded currently selected set of active processors is automatically restored to its original state.

This form should be used wherever evaluation of the forms in *body* might cause control flow to abnormally pass out of a *Lisp form that restricts the currently selected set (for example, by a call to **throw, return, return–from,** or **go** within a ***when** form). The **with–css–saved** macro uses an **unwind–protect** to trap such non-local exits and restore the currently selected set.

EXAMPLES

The following function definitions demonstrate the use of **with–css–saved**. Both functions return the result of dividing **y** by **x** in all processors where **y** > 0. If any value of **x** is zero, both functions return **nil**.

```
(defun css-not-preserved (x y)
  (block exit
    (*when (>!! y (!! 0))
      (if (*or (zerop!! x))
          (return-from exit nil)
          (/!! y x)
          ))))

(defun css-preserved (x y)
  (block exit
    (with-css-saved
      (*when (>!! y (!! 0))
        (if (*or (zerop!! x))
            (return-from exit nil)
            (/!! y x)
            )))))
```

The difference between the functions lies in the way **css–preserved** uses the **with–css–saved** macro around its conditional to restore the currently selected set. For example, given the configuration defined by

```
(*cold-boot :initial-dimensions '(512))
```

the expression

```
(*all (progn   (css-not-preserved (!! 0) (self-address!!))
               (*sum (!! 1))))
```

returns 511.

The pvar returned by **(self–address!!)** is 0 in processor zero, so **css–not–preserved** deselects processor 0. When the call to **return–from** in **css–not–preserved** is executed because **x** contains the value 0 in every processor, **css–not–preserved** does nothing to restore the currently selected set, leaving processor 0 deselected.

The expression

```
(*all (progn   (css-preserved (!! 0) (self-address!!))
               (*sum (!! 1))))
```

returns 512. By enclosing the ***when** conditional with the **with–css–saved** macro, the **css–preserved** function ensures that the currently selected set is automatically restored when the call to **return–from** is executed.

NOTES

For the purposes of forms that execute non-local exits, the **with–css–saved** macro is functionally equivalent to a call to **unwind–protect**. When a non-local exit is performed, an **unwind–protect** is executed to restore the currently selected set, and then the exit continues normally. Evaluation does *not* continue with the form immediately following the **with–css–saved**. For example, when

```
(catch 'exit
   (with-css-saved
      (yin data-pvar)
      (when win-yin
         (throw 'exit nil)))
   (yang data-pvar))
```

is evaluated, if the variable **win–yin** has the value **t**, then **(yin data–pvar)** is evaluated, but **(yang data–pvar)** is not.

REFERENCES

See also the processor selection operators

***all**	
***case**	**case!!**
***cond**	**cond!!**
***ecase**	**ecase!!**
***if**	**if!!**
***unless**	***when**

with–processors–allocated–for–vp–set [*Macro*]

Temporarily instantiates (assigns a geometry to) a flexible VP set for the duration of a set of body forms.

SYNTAX

with–processors–allocated–for–vp–set (*vp–set* **&key :dimensions :geometry**)
&body *body*

ARGUMENTS

vp–set	Flexible VP set. Virtual processor set defined with **def–vp–set**.
:dimensions	Integer list or **nil**. Size of dimensions with which to instantiate *vp–set*. Must be **nil** if *geometry* argument is supplied.
:geometry	Geometry object, obtained by calling the function **create–geometry**. Defines geometry of *vp–set*.
body	*Lisp forms. Body forms to be evaluated with *vp–set* instantiated.

RETURNED VALUE

body–value	Scalar or pvar value. Value of final form in *body*.

SIDE EFFECTS

Temporarily defines geometry of *vp–set* and allocates any associated pvars, for the duration of the *body* forms, then deinstantiates *vp–set* and deallocates any associated pvars.

DESCRIPTION

This macro expands into a form that instantiates *vp–set* by a call to **allocate–processors–for–vp–set**, using the supplied *dimensions* or *geometry* as arguments. As with the **allocate–processors–for–vp–set** function, one or the other of the **:dimensions** or **:geometry** arguments may be supplied, but not both. The form then executes the supplied *body* forms and finally calls **deallocate–processors–for–vp–set** to deinstantiate *vp–set*.

EXAMPLES

A sample call to **with–processors–allocated–for–vp–set** is

```
(def-vp-set my-vp-set nil
            :*defvars '((value-pvar (self-address!!))))

(with-processors-allocated-for-vp-set (my-vp-set
                                       :dimensions '(32 32 32))
   (*with-vp-set my-vp-set
      (*set value-pvar (*!! value-pvar (!! 2)))
      (ppp value-pvar :end 8)))

0 2 4 6 8 10 12 14
```

The following example shows how a flexible VP set can be used repeatedly to process a set of data files. In the example, a single flexible VP set is used, which is instantiated and deinstatiated once for each file in such a way that it is just large enough to hold each file's data.

```
(def-vp-set file-data-vp-set nil
            :*defvars '((file-data-pvar)))

(dolist (file files-to-be-processed)
  (let ((file-size (get-file-size file)))
    (with-processors-allocated-for-vp-set file-data-vp-set
        :dimensions (next-power-of-two->= file-size)
      (*with-vp-set file-data-vp-set
          (*set file-data-pvar (read-file-data!!))
          (process-file-data file-data-pvar)))))
```

REFERENCES

See also the following flexible VP set operators:

allocate–vp–set–processors
deallocate–vp–set–processors
set–vp–set–geometry
allocate–processors–for–vp–set
deallocate–processors–for–vp–set

*with–vp–set [*Macro*]

Dynamically binds the supplied VP set as the current VP set for the duration of the supplied body forms.

SYNTAX

***with–vp–set** *vp–set* **&body** *body*

ARGUMENTS

vp–set	VP set object. VP set to be made current. Must be defined and instantiated.
body	*Lisp forms. Body forms to be evaluated.

RETURNED VALUE

body–value	Scalar or pvar value. Value of final form in *body.*

SIDE EFFECTS

Temporarily changes the current VP set to *vp–set* during the evaluation of the supplied *body* forms.

DESCRIPTION

This macro is used to temporarily switch VP sets for the duration of a section of code.

The currently selected VP set is dynamically scoped. The ***with–vp–set** form temporarily binds the current VP set to *vp–set.* Thus, while a ***with–vp–set** form is executing, the global variables related to VP sets are dynamically bound according to the size, shape, and properties of *vp–set.*

The following global variables are affected when the current VP set is changed:

current–cm–configuration	***current–grid–address–lengths***
current–send–address–length	***current–vp–set***

log-number-of-processors-limit ***number-of-dimensions***
number-of-processors-limit ***ppp-default-end***
ppp-default-start **t!!** **nil!!**

EXAMPLES

Each VP set maintains its own currently selected set of processors. Nested calls to ***with-vp-set** that switch between VP sets also switch between the currently selected sets maintained by the VP sets. This is illustrated by the example shown below.

```
(def-vp-set fred '(1024 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!!)))
                (setq zero-off (*sum (!! 1)))   ;16,383 VP's

            (setq zero-on (*sum (!! 1))))))      ;16,384 VP's
   (*sum (!! 1)))  =>  32768

zero-off  =>  16383
zero-on   =>  16384
```

When a VP set is created, it is defined to have all processors selected, so the initial call to ***with-vp-set fred** selects the **fred** VP set with all virtual processors active. The first ***when** statement reduces the number of active processors in **fred** by half by selecting only even-numbered processors, and the call to ***with-vp-set anne** selects the **anne** VP set, which has 262,144 virtual processors.

The second invocation of ***with-vp-set fred** reselects the **fred** VP set with the same currently selected set as before: only processors of even-numbered addresses are active. The second call to ***when** further restricts the selected set of **fred** by deactivating processor 0. Inside this ***when** statement, a call to **(*sum (!! 1))** returns 16383, the number of active processors in **fred**. The call to **(*sum (!! 1))** immediately following the ***when** returns 16384, the number of active processors in **fred** with processor 0 included.

When execution passes back into the ***with–vp–set** form that originally selected the **fred** VP set, all processors are again active and **(*sum (!! 1))** returns 32768, the total number of virtual processors in **fred**.

If the body of a call to ***with–vp–set** must be evaluated with all processors selected, rather than only those processors currently active in the selected VP set, it should be surrounded by a call to ***all**, as in

```
(*with-vp-set fred
   (*all
      (*set x (-!! y x))))
```

REFERENCES

See also the related operation
set–vp–set

*xor [*Defun]

Takes the logical XOR of all values in a pvar, returning a scalar value.

SYNTAX

***xor** *pvar–expression*

ARGUMENTS

pvar–expression	Pvar expression. Pvar to which global XOR is applied.

RETURNED VALUE

xor–scalar	Scalar boolean value. The logical XOR of the values of *pvar–expression* in all active processors, i.e., the value **t** if an odd number of the values are non-**nil**, and the value **nil** otherwise.

SIDE EFFECTS

None.

DESCRIPTION

The ***xor** function is a global operator. It takes the logical XOR of all values in a pvar, returning a scalar value. Effectively, ***xor** treats the value of *pvar–expression* in all active processors as a set of boolean values. It returns the value **t** if an odd number of those values are non-**nil**, and returns the value **nil**.

If there are no active processors, this function returns **nil**.

EXAMPLES

```
(*xor t!!)         =>  NIL      ;;; t in all processors
(*xor nil!!)       =>  NIL      ;;; t in no processors

;;; t in every other processor
(*xor (evenp!! (self-address!!)))  =>  NIL
(*xor (oddp!! (self-address!!)))   =>  NIL

;;; t in every third processor (an odd number)
(*xor (zerop!! (mod!! (self-address!!) (!! 3))))  =>  T

;;; an example using non-boolean values
(*xor (if!! (zerop!! (self-address!!))
            nil!!
            (self-address!!))) => T  ;;; All but one non-NIL
```

REFERENCES

See also the related global operators:

***and**	***integer-length**	***logand**
***logior**	***logxor**	***max**
***min**	***or**	***sum**

See also the related logical operators:

and!! **not!!** **or!!** **xor!!**

xor!! [*Function*]

Performs a parallel logical XOR operation in all active processors.

SYNTAX

xor!! &rest *pvar–exprs*

ARGUMENTS

pvar–exprs Pvar expressions. Pvars to which parallel XOR is applied.

RETURNED VALUE

xor–pvar Temporary boolean pvar. Contains in each active processor the logical XOR of the corresponding values of the *pvar–exprs*. If no *pvar–exprs* are given then **nil!!** is returned.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

This performs the XOR function on all the *pvar–exprs*. If no *pvar–exprs* are given then **nil!!** is returned. In each processor, **xor!!** returns **t** if an odd number of the supplied *pvar–exprs* have the value **t** in that processor, and otherwise returns **nil**.

EXAMPLES

```
(xor!!    (evenp!! (self-address!!))
          (oddp!! (self-address!!)))  <=>  t!!

(ppp (xor!!  (self-address!!)
             (evenp!! (self-address!!)))
     :end 8)
NIL T NIL T NIL T NIL T
```

REFERENCES

See also the related global operators:

***and**	***integer–length**	***logand**
***logior**	***logxor**	***max**
***min**	***or**	***sum**
***xor**		

See also the related logical operators:

and!!	**not!!**	**or!!**

zerop!! [*Function*]

Performs a parallel test for zero values on the supplied pvar.

SYNTAX

zerop!! *numeric–pvar*

ARGUMENTS

numeric–pvar	Numeric pvar. Tested in parallel for zero values.

RETURNED VALUE

zerop–pvar	Temporary boolean pvar. Contains the value **t** in each active processor where the corresponding value of *numeric–pvar* is zero. Contains **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

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

EXAMPLES

```
(zerop!! (mod!! (self-address!!) (!! 2)))
  <=>
(evenp!! (self-address!!))
```

!! [Function]

Returns a temporary pvar with the same value in each active processor.

SYNTAX

!! *scalar–expression*

ARGUMENTS

scalar–expression	Scalar expression. The value to be stored in each processor of the returned pvar. The data type of *scalar–expression* must be either a number, a character, an array, or a structure.

RETURNED VALUE

constant–pvar	A temporary pvar with the value of *scalar–expression* in each active processor.

SIDE EFFECTS

Allocates the new temporary pvar on the stack.

DESCRIPTION

The *Lisp function **!!** returns a temporary pvar containing the value of *scalar–expression* in each active processor. The *scalar–expression* must be a number, a character, an array, or a structure.

Note: The original purpose of **!!** was to allow you to provide constant pvar arguments to *Lisp functions, as in the expression

```
(+!! (!! 2) (!! 3) (!! 4))
```

*Lisp functions now allow you to pass scalar constants directly (the call to !! to convert them to pvars is made automatically by *Lisp itself). This means that you will rarely ever have to use the **!!** function yourself.

If *scalar–expression* evaluates to an array, a complete copy of the array is stored in each active processor. If the array has a fill pointer, it is ignored; all elements of the array are copied into the CM. Adjustable arrays are copied and stored as fixed-size arrays. Displaced arrays are copied and stored as non-displaced arrays. The data type of the returned pvar depends on the data types of the elements in the array. If the array contains elements of various types, the *Lisp rules of type coercion apply.

If *scalar–expression* evaluates to a scalar structure object (of a structure type defined by a call to ***defstruct**) an **equalp** copy of the object is stored in each active processor of the returned pvar.

EXAMPLES

By distributing a single scalar value to all processors, the **!!** function provides the same functionality in *Lisp as scalar values provide in Common Lisp (see Figure 6).

A typical call to **!!** is very simple.

```
(!! 5)    ;;; Returns a pvar with 5 in each processor
```

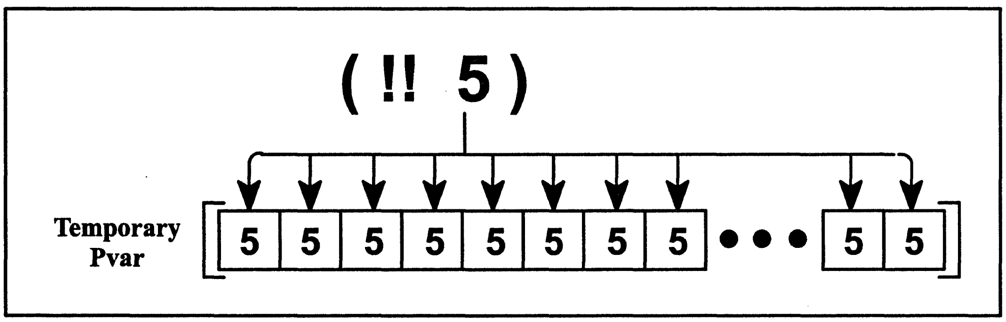

Figure 6. The expression **(!! 5)** distributes a scalar value (5) to all processors.

In *Lisp, **!!** is most often used to pass a constant value to a function, as in

```
(random!! (!! 10))
```

The function **random!!** expects a single pvar argument whose value in each processor is the upper bound of the random number to be calculated in that processor. The above example returns a temporary pvar containing a random value between 0 and 9 in each processor. Note that this differs from

```
(random!! (1+!! (self-address!!)))
```

which returns a pvar whose value in each processor is a random number between 0 and the processor's send address. Here, the pvar argument has a different value in every processor.

As the following example demonstrates, **!!** is very useful in comparisons.

```
(<!! (self-address!!) (!! 256))
```

This returns a pvar with **t** in each processor whose send address is less than 256, and **nil** in all other active processors.

The following is a call to **!!** with an array argument:

```
(*defvar parallel-array (!! #(1 2 3)))

(ppp parallel-array)
#(1 2 3) #(1 2 3) #(1 2 3) . . . #(1 2 3) #(1 2 3)

(setq *print-array* t)
(pref parallel-array 1) => #(1 2 3)
```

This creates a pvar with a copy of the array **#(1 2 3)** in each processor. Using **pref**, the copy of the array in each processor is accessed. Individual elements of the parallel arrays may be accessed using **aref**.

Nested arrays of arbitrary depth are legal arguments to **!!**. For instance, an array of arrays is a permissible argument to **!!**. The expression

```
(!!   #( #(2 4) #(6 12) #(7 16) #(5 20) #(2 56) ))
```

creates a pvar with an array of arrays in each processor. Calling **!!** with nested arrays can be a very slow operation.

An example using structures is

```
(*defstruct elephant
   (wrinkles 30000 :type (unsigned-byte 16))
   (tusks t :type boolean))

(!! (make-elephant :wrinkles 0 :tusks nil))
```

This creates a pvar with a wrinkle-free, tuskless **elephant** in each processor.

NOTES

It is an error to call **!!** with an array containing elements that cannot, according to the *Lisp rules of type coercion, be coerced into a single, fixed-size type. For example,

```
(!! #(1 2 3 #\e #\r #\r #\o #\r #\!))
```

is in error because the array argument contains both integers and characters.

Implementation Note:

In Lucid and Sun Common Lisp versions of *Lisp, front-end floating-point numbers are always stored as double-precision numbers, regardless of thier actual precision. This means that the expression

```
(!! 3.14)
```

is ambiguous—there's no way to tell whether you intended to create a single-precision or a double-precision floating-point pvar, even if you declare the returned type of the **!!** expression!

For this reason, *Lisp has an internal variable, ***lisp-i::*default-float-precision***, that specifies the "default" precision of an ambiguous floating-point **!!** expression. This variable can be set to either **:single** or **:double**, and defaults to **:single**.

This only affects the *Lisp interpreter. The *Lisp compiler has more information about the types of values in these expressions, so compiled code doesn't have this problem.

REFERENCES

See also the pvar allocation and deallocation operations

allocate!!	**array!!**	
***deallocate**	***deallocate-*defvars**	***defvar**
front-end!!	***let**	***let***
make-array!!	**typed-vector!!**	**vector!!**

=!!, /=!!, <!!, <=!!, >!!, >=!! [*Function*]

Perform parallel numerical comparisons on the supplied pvar arguments.

SYNTAX

=!!, /=!!, <!!, <=!!, >!!, >=!! *numeric–pvar* **&rest** *numeric–pvars*

ARGUMENTS

numeric–pvar, numeric–pvars Pvars to be compared.

RETURNED VALUE

These functions each return a single temporary boolean pvar, as described below:

equal–pvar The value **t** in each active processor where the *numeric–pvar* arguments are equal, and **nil** in all other active processors.

not–equal–pvar The value **t** in each active processor where the *numeric–pvar* arguments are ~~equal~~ not equal, and **nil** in all other active processors.

less–than–pvar The value **t** in each active processor where the *numeric–pvar* arguments are ~~equal~~ <, and **nil** in all other active processors.

not–greater–pvar The value **t** in each active processor where the *numeric–pvar* arguments are ~~equal~~ <=, and **nil** in all other active processors.

greater–pvar The value **t** in each active processor where the *numeric–pvar* arguments are ~~equal~~ >, and **nil** in all other active processors.

not–less–pvar The value **t** in each active processor where the *numeric–pvar* arguments are ~~equal~~ >=, and **nil** in all other active processors.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

These functions perform parallel comparisons; each function returns a temporary pvar that contains **t** in each active processor where the argument pvars pass the corresponding relational test (equality, less-than, greater-than, etc.), and **nil** in all other active processors. These functions provide the same functionality for numeric pvars as the Common Lisp operators **=**, **/=**, **<**, **<=**, **>**, and **>=** provide for numeric scalars.

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

EXAMPLES

These functions can be used to compare the values of a pvar with some constant value. For example, if **numeric-pvar** contains the values 0, 5, 1, –4, 5, etc., then the pvar returned by

```
(=!! numeric-pvar (!! 5))
```

contains the values **nil**, **t**, **nil**, **nil**, **t**, etc.

Similarly, one pvar can be compared with another. The expression

```
(<!! numeric-pvar (self-address!!))
```

returns a pvar with the value **t** in each processor for which **numeric-pvar** is less than the processor's send address.

These functions are especially useful in combination with the processor selection operators. For example,

```
(*when (>!! data-pvar (!! 10))
   (*set data-pvar (*!! data-pvar (!! 2))))
```

multiplies **data-pvar** in processors where **data-pvar** is greater than 10. The macro ***when** is used with **>!!** to select processors where **data-pvar** is greater than 10. The value of **data-pvar** in those processors is multiplied by 2 using ***!!** and stored back into **data-pvar** by ***set**.

NOTES

An error is signalled if any of the *numeric-pvar* arguments contains a non-numeric value in any active processor.

+!!, –!!, *!!, /!! *[Function]*

Perform parallel addition, subtraction, multiplication, or division on the supplied pvars.

SYNTAX

+!!, *!! **&rest** *numeric–pvars*
–!!, /!! *numeric–pvar* **&rest** *numeric–pvars*

ARGUMENTS

numeric-pvar, numeric–pvars Numeric pvars to be combined arithmetically.

RETURNED VALUE

result–pvar Temporary numeric pvar. In each active processor, contains the result of the arithmetic operation on the *numeric–pvars*.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

These functions provide the same functionality for numeric pvars as the Common Lisp arithmetic operations **+**, **–**, *****, and **/** provide for numeric scalars. Each function performs an arithmetic operation on the supplied *numeric–pvars*.

The **+!!** function performs parallel addition, returning **(!! 0)** when no arguments are supplied. The ***!!** function performs parallel multiplication, returning **(!! 1)** when no arguments are supplied.

The **–!!** function performs parallel subtraction, or negation, if only one argument is supplied. The **/!!** function performs a parallel division, or inversion, if only one argument is supplied.

Note: Both **–!!** and **/!!** require at least one numeric-pvar argument. Also, since *Lisp lacks "rational number pvars", **/!!** always returns a floating-point or complex pvar.

EXAMPLES

The function **+!!** can be used to increment a pvar by some constant value. For example,

```
(+!! numeric-pvar (!! 5))
```

returns a pvar whose value in each processor is the value of **numeric–pvar** plus 5.

Similarly, **–!!** can be used to find the difference of several pvars. The expression

```
(-!! particles-pvar protons-pvar neutrons-pvar)
```

returns a temporary pvar containing in each processor the result of subtracting **protons–pvar** and **neutrons–pvar** from **particles–pvar** in that processor.

The ***!!** operator can be used together with the processor selection operators to modify the values of a selected group of processors. For example,

```
(*when (>=!! baggage-weight-pvar (!! 150))
   (*set passenger-charge-pvar
      (*!! current-rate-pvar (!! 2))))
```

uses ***!!** to change the fare for passengers with excess baggage. The macro ***when** is used with **>=!!** to select those processors in which **baggage–weight–pvar** is greater than or equal to 150. In these processors, ***!!** is used with ***set** to store twice the value of **current–rate–pvar** in **passenger–charge–pvar**.

NOTES

For **/!!**, if there is only one *numeric–pvar* argument, it is an error if the pvar has the value 0 in any active processor. If there is more than one argument, it is an error if any *numeric–pvar* other than the first argument has the value 0 in any active processor.

An error is signalled if any of the *numeric–pvar* arguments contains a non-numeric value in any active processor.

If the data types of the argument pvars differ, the *Lisp rules of type coercion apply.

1+!! [*Function*]

Performs parallel addition/subtraction of 1 to/from the supplied pvar.

SYNTAX

1+!! *numeric–pvar*
1–!! *numeric–pvar*

ARGUMENTS

numeric–pvar	Numeric pvar. Incremented or decremented in parallel.

RETURNED VALUE

increment–pvar	Temporary numeric pvar. In each active processor, contains a copy of the value of *numeric–pvar* incremented or decremented by one.

SIDE EFFECTS

The returned pvar is allocated on the stack.

DESCRIPTION

The **1+!!** function performs a parallel increment, and the **1–!!** function performs a parallel decrement. Both functions return a copy of the *numeric–pvar* with values either incremented or decremented by 1. These functions provide the same functionality for numeric pvars as the Common Lisp functions **1+** and **1–** provide for numeric scalars.

EXAMPLES

The **1+!!** function is a contraction of the expression

```
(+!! numeric-pvar (!! 1))
```

and performs identically.

The **1–!!** function is a contraction of the expression

```
(-!! numeric-pvar (!! 1))
```

and performs identically.

NOTES

An error is signalled if the *numeric–pvar* argument contains a non-numeric value in any active processor.

REFERENCES

The function ***incf** can be used to destructively increment its argument pvar. See the dictionary entry on ***incf** for more information.

The function ***decf** can be used to destructively decrement its argument pvar. See the dictionary entry on ***decf** for more information.