The
Connection Machine
System

*Lisp Release Notes

Version 5.1
June 1989

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, June 1989

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

Contents

About These Release Notes

Objectives

The **Lisp Release Notes Version 5.1* are published to inform *Lisp programmers about all new and changed *Lisp features introduced with the Connection Machine System Software version 5.1.

Intended Audience

The reader is assumed to have a working knowledge of Common Lisp, as described in *Common Lisp: The Language*, and of *Lisp, as described in the *Lisp documentation for version 5.0. The reader is also assumed to have a general understanding of the Connection Machine system. The volume entitled *Connection Machine Front-End Subsystems* provides the necessary background information about the Connection Machine system.

Revision Information

These release notes are new with *Lisp version 5.1. They do not replace **Lisp Release Notes Version 5.0*, nor do they replace any other manual in the *Lisp documentation for version 5.0.

Organization of These Release Notes

1 About Version 5.1
Identifies *Lisp and version 5.1.

2 Porting Code to Version 5.1,
Explains what to do to ensure that 5.0 *Lisp code runs under 5.1.

3 *Lisp Language Version 5.1
Describes language features that are new and enhanced in version 5.1.

4 *Lisp Compiler Version 5.1
Describes compiler features that are new and enhanced in version 5.1.

5 *Lisp Simulator Version 5.1
Describes simulator features that are new and enhanced in version 5.1.

Related Manuals

- **Lisp Release Notes Version 5.0*

  The version 5.0 release notes provide a succinct overview of the many new features introduced in version 5.0 and of the changes made to *Lisp between the release of version 4.3 and the release of version 5.0. These are essential reading.

- *Supplement to the *Lisp Reference Manual Version 5.0*

  This manual updates the **Lisp Reference Manual*, adding descriptions of all features new with the release of *Lisp version 5.0.

- **Lisp Compiler Guide Version 5.0*

  This manual describes the current implementation of the *Lisp compiler.

- *Connection Machine Front-End Subsystems*

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

- *Connection Machine Parallel Instruction Set*

  The **Lisp Reference Manual* explains how to call Paris from *Lisp. Users who wish to do so should refer also to the Paris manual.

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

  This book defines the de facto industry standard for the Common Lisp language.

Notation Conventions

The notation conventions used in these release notes are the same as those used in all current *Lisp documentation.

Convention	Meaning
boldface	*Lisp language elements, such as keywords, operators, and function names, when they appear embedded in text.
italics	Parameter names and placeholders in function formats.
`typewriter`	Code examples and code fragments.
% **boldface** `typewriter`	In interactive examples, user input is shown in **boldface** and system output is shown in `typewriter` font.

Customer Support

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

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

To contact Thinking Machines Customer Support:

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

For Symbolics users only:

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

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

Please supplement the automatic report with any further pertinent information.

1 About Version 5.1

The *Lisp language is a parallel extension of Common Lisp for programming the Connection Machine system. Programs using *Lisp typically include both Common Lisp and *Lisp constructs. *Lisp provides three separate utilities: the interpreter, which extends the Common Lisp interpreter; the compiler, which extends the Common Lisp compiler; and the simulator, which simulates a Connection Machine running *Lisp.

Version 5.1 is an incremental *Lisp release. It provides an expanded range of parallel Lisp operations and corrects a number of implementation errors. *Lisp version 5.1 does not, however, differ significantly in either functionality or design from *Lisp version 5.0.

2 Porting Code to Version 5.1

*Lisp code written under version 5.0 runs under version 5.1 unchanged with the following exceptions:

- *Lisp version 5.1 requires Lucid Common Lisp on Sun-4 front ends. For systems with Sun-4 front ends, some changes need to be made to Lisp code and to the front-end development environment in order to move to Lucid version 3.0.
- Calls to **random!!** that take floating-point data produce different results under the two versions. A workaround is documented below.
- Data stored on a DataVault under version 5.0 is not guaranteed to have the same VP set geometry grid pattern when read into a CM under version 5.1.
- Code containing obsolete functions is not guaranteed to run.

2.1 Lucid Common Lisp Version 3.0

On Sun-4 front ends only, CM System Software version 5.1 runs under Lucid Common Lisp version 3.0. Lucid 3.0 is significantly different from Lucid 2.1. *Lisp programmers are advised to obtain Lucid 3.0 documentation. The Lucid changes that most affect *Lisp programs are noted below.

(1) **Name changes**

The naming convention for certain Lucid functions has changed. Functions whose names began with **SYS::** in Lucid version 2.1 begin with **LCL::** in version 3.0. For example, **(SYS::quit)** is now **(LCL::quit)**.

(2) **Use of the change-memory-management function discouraged**

The **change-memory-management** function is not recommended by Lucid for version 3.0. If it is nonetheless used, the **:growth-limit** keyword valued should be much less than previously recommended by Thinking Machines Corporation. Failing to reduce the **:growth-limit** value before using **change-memory-management** causes problems with the garbage collection mechanism.

Here is an acceptable **change-memory-management** call used by some Thinking Machines Corporation developers in their **lisp-init.lisp** files.

```
(change-memory-management
   :expand 64
   #+sun :growth-limit #+sun 768
   )
```

(3) **Foreign function interface changed**

The Lucid foreign function interface has changed. Please consult the Lucid Common Lisp version 3.0 documentation for details.

(4) **Two Lucid compiler modes: production and development**

Lucid Common Lisp version 3.0 supports two modes of compiling: production and development. The production compiler is an optimizing compiler; it compiles more slowly but produces more efficient code than the development compiler. The development compiler is a non-optimizing compiler that compiles very rapidly. The development compiler always uses full Lucid safety checking.

NOTE

The Lucid 3.0 compiler is completely independent of the *Lisp compiler with regard to options such as safety. The *Lisp compiler has its own, independent, safety setting.

The *Lisp compiler translates *Lisp code into Common Lisp code with calls to Paris. Then the Lucid Common Lisp compiler translates the Lisp code generated by the *Lisp compiler into native machine instructions.

See the **Lisp Compiler Guide* for more information about the *Lisp compiler. Refer to Lucid 3.0 documentation for more information about the Lucid Common Lisp compiler and its production and development modes.

To switch easily between the Lucid compiler production and development modes, place the following two function definitions in your **lisp-init.lisp** file:

```
(defun prod ()
   (proclaim '(optimize (compilation-speed 0)
                        (safety 1) (speed 3))))

(defun def ()
   (proclaim '(optimize (compilation-speed 3)
                        (safety 3) (speed 2))))
```

(These settings are taken from the Lucid 3.0 documentation.)

When developing code interactively, make the development compiler the default by placing the expression **(dev)** in your **lisp-init.lisp** file, immediately after the two function definitions. Using the development compiler can significantly speed up the compilation process.

To compile developed code for distribution, enable the production compiler mode by typing **(prod)** at top level.

(5) **Lucid ephemeral garbage collector**

In previous Lucid releases, garbage collection occurred frequently and took significant amounts of time. Lucid Common Lisp version 3.0 includes an ephemeral garbage collector. Consequently, full garbage collection is neither as frequent nor as noticeable.

2.2 CM Data Storage Format Changed

Given an identically sized machine and identical VP set dimensions, a geometry created when a VP set is created is not necessarily identical under *Lisp versions 5.0 and 5.1. If data from VP set *A* is written to the DataVault under version 5.0 and subsequently read into VP set *A* under version 5.1, the data is not guaranteed to be stored in the Connection Machine in the same grid order.

2.3 Floating-Point random!! Results

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

In *Lisp version 5.1, **random!!** produces a different sequence of floating-point pseudo-random numbers than it did in version 5.0. Random integer sequences are identical between version 5.0 and version 5.1.

To make **random!!** produce the exact same sequence of pseudo-random floating-point numbers as it did in version 5.0, execute the following form at top level:

```
(setq slc::*f-random-function* 'slc::float-random)
```

2.4 Obsolete Language Features

The following experimental *Lisp vector functions are now obsolete:

sf-v+!!
sf-v-!!
sf-v*!!
dsf-v+!!
dsf-v-!!
dsf-v*!!
sf-dot-product!!

These functions were originally documented in Chapter 10 of the *Supplement to the *Lisp Reference Manual* version 5.0. They were introduced to provide optimized interpreter performance for floating-point vector operations. The *Lisp compiler now compiles the **v+!!**, **v-!!**, **v*!!**, and **dot-product!!** functions. Fast interpreted floating-point vector operations are therefore now unnecessary.

3 *Lisp Language Version 5.1

Several new *Lisp functions and a number of enhancements are included in *Lisp version 5.1.

3.1 New *Lisp Language Features

3.1.1 Overview of New Language Features

The following categories of *Lisp operations have new capabilities that distinguish version 5.1 of the *Lisp language from version 5.0:

- *Grid Communication.* The new operations **news-direction!!**, ***news-direction**, and **off-grid-border-relative-direction-p!!** allow communication along an arbitrary NEWS axis, without requiring any indication of how many dimensions exist in the current VP set.

- *VP sets.* A number of new features for manipulating VP sets are included in version 5.1. One, **set-vp-set-geometry**, changes the geometry of a VP set. Another, **with-processors-allocated-for-vp-set**, temporarily allocates processors for a VP set. Three new functions return state information about a VP set: **vp-set-rank**, **vp-set-total-size**, and **vp-set-vp-ratio**. Finally, **deallocate-processors-for-vp-set** is added in 5.1, providing a way to undo the effects of **allocate-processors-for-vp-set**, which was introduced in 5.0.

- *Computation.* The new **amap!!** function maps a given parallel operation over any number of supplied array pvars.

- *Macro Unbinding.* The new **un*defun** function unbinds the macro and function definitions for any number of provided symbols that have previously been defined with ***defun**.

- *Aggregation.* The new **array!!** and **vector!!** functions aggregate any number of provided pvars into either an array pvar or a vector pvar.

- *Array Referencing.* The new functions **row-major-aref!!** and **row-major-sideways-aref!!** make it possible to index an array pvar independent of dimensionality. They treat each component array as a vector that is laid out in row major order.

- *Indirect Addressing.* The functions ***slicewise** and ***processorwise** convert array pvars between a slicewise (sideways) addressing orientation and the normal, processorwise orientation. Functionally, these replace ***sideways-array**, introduced in 5.0.

  Still supported, ***sideways-array** toggles array pvar addressing between the sideways (slicewise) and processorwise modes. Newly documented with 5.1, **sideways-array-p** tests for the current mode. In a future release, ***sideways-array** is likely to become obsolete.

- *Debugging.* The new ***trace** and ***untrace** functions make it possible to trace macros defined with ***defun**. The experimental ***room** function prints information about available CM memory.

3.1.2 Description of New Language Features

The following *Lisp operations are new with version 5.1:

amap!!
array!!
deallocate-processors-for-vp-set
news-direction!!
***news-direction**
off-grid-border-relative-direction-p
***processorwise**
***room**
row-major-aref!!
row-major-sideways-aref!!
set-vp-set-geometry
sideways-array-p
***slicewise**
***trace**
un*defun
vector!!
vp-set-deallocated-p
vp-set-rank
vp-set-total-size
vp-set-vp-ratio
with-processors-allocated-for-vp-set

A description of each follows.

amap!!

amap!! *operator array-pvar* **&rest** *array-pvars* [*Function*]

Applies the specified function to each element of the supplied array pvars and returns the result as an array pvar.

The *operator* parameter may be any parallel function capable of combining the number of array pvars supplied.

The *array-pvar* parameter is required. Any number of **&rest** *array-pvars* parameters may be supplied. All supplied array pvars must have the same dimensions.

The function **amap!!** applies *operator* to each element of each array pvar. In this way, sets composed of one element from each array pvar are combined. The result is an array pvar, each element of which represents the result of one application of *operator* over the corresponding elements of the supplied array pvars.

The function **amap!!** is similar to the Common Lisp function **map**. However, while **map** works only on vectors, **amap!!** works on any type of array. Also, **amap!!** requires no result type specification.

The function **amap!!** is similar to the *Lisp function ****map**, introduced in 5.0. The syntax is different for the two functions and **amap!!** returns a pvar result while ***map** is executed for side effect.

The **amap!!** function removes the need for experimental sequence functions such as **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))))))
)
```

array!!

array!! *dimensions* **&rest** *pvars* [*Function*]

Creates an array pvar with the specified dimensions and containing the specified pvars.

The function **array!!** takes any number of pvar expressions. The number of array elements specified by *dimensions* must match the number of pvars supplied.

An array pvar is returned. Each element of this array pvar is an array that contains one element of each supplied pvar, taken in row-major order.

For example,

```
(pref (array!! '(2 2) (!! 0) (!! 1) (!! 2) (!! 3)) 25)

=> #2A((0 1) (2 3))
```

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.)

---

deallocate-processors-for-vp-set

deallocate-processors-for-vp-set *vp-set* [*Function*]
&key :ok-if-not-instantiated

Deallocates all processors previously allocated for the specified VP set if those processors were allocated by a call to **allocate-processors-for-vp-set**.

The *vp-set* parameter must be a 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 deallocated and its *defvars are not destroyed. However, any pvar belonging to *vp-set* and created using **allocate!!** is deallocated 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 an error is signaled if the provided VP set is not instantiated at the

time of the call. If the value is **nil**, then an error is signaled if *vp-set* is uninstantiated. If **:ok-if-not-instantiated** is set to **t**, no error is signaled if *vp-set* is uninstantiated.

Usage Note: This function may also be used under the name **deallocate-vp-set-processors**, which corresponds to the undocumented function name **allocate-vp-set-processors.**

---

news-direction!!

news-direction!! *source-pvar dimension-scalar distance-scalar* [*Macro*]

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. (See the reference entry for **news!!** on page 84 of the *Supplement to the *Lisp Reference Manual* version 5.0.)

The result is returned as a pvar of the same type as *source-pvar.*

The *source-pvar* parameter must be in the current VP set. Data is copied from this pvar into the result pvar.

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,

```
(news-direction!! my-pvar 1 2)
<=>
(news!! my-pvar 0 2 0)
```

assuming a three-dimensional machine configuration.

---

*news-direction

***news-direction** *source-pvar destination-pvar dimension-scalar distance-scalar* [*Defun*]

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*. (See the ***news** reference entry on page 85 of the *Supplement to the *Lisp Reference Manual* version 5.0.)

This operation is executed for side effect; the result is stored in the *destination-pvar*.

The *source-pvar* and *destination-pvar* parameters must both be in the current VP 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,

```
(*news-direction my-pvar my-result 2 3)
<=>
(*news my-pvar my-result 0 0 3)
```

assuming a three-dimensional machine configuration.

---

off-grid-border-relative-direction-p

off-grid-border-relative-direction-p!! *dimension-scalar distance-scalar* [*Function*]

Tests the relative grid addresses indicated by the specified dimension and distance 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**.

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,

```
(off-grid-border-relative-direction-p!! 1 5)
<=>
(off-grid-border-relative-p!! 0 5 0)
```

assuming a three-dimensional machine configuration.

---

*processorwise

***processorwise** *array–pvar* [*Defun*]

Converts a slicewise (sideways) array to the normal, processorwise orientation.

The *array–pvar* parameter must be a sideways (slicewise) array, otherwise an error is signaled.

There are restrictions on arrays that can be turned sideways. See the reference entry for ***sideways-array** on page 33 of the *Supplement to the *Lisp Reference Manual* version 5.0.

Note that ***processorwise** is equivalent to ***sideways-array** when ***sideways-array** is called on an array that is already sideways (slicewise).

---

*room

> ** ** Experimental ** **
>
> *This function is experimental. *Lisp programmers are welcome to try it at their own risk and with the understanding that, if insufficiently popular, it may not be supported in future releases.*

***room &key :how :print-statistics :stream** [*Function*]

Collects and prints information about CM memory use.

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 bytes of CM memory allocated on the *Lisp stack.
- The second return value reports the total number of bytes of CM memory occupied by temporary pvars allocated with **allocate!!**.
- The third return value reports the total number of bytes of CM memory occupied by pvars created using ***defvar**.
- The fourth return value reports the total number of bytes of CM memory in use as overhead, including overhead for the *Lisp VP mechanism and overhead for Paris.

The **:how** keyword argument specifies how memory information is collected and printed. This 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 the 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 **: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.

row-major-aref!!

row-major-aref!! *array-pvar row-major-index-pvar* [*Function*]

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.

Consider the following code, for example:

```
(pref (row-major-aref!! (!! #2A((5 8) (3 0))) (!! 2)) 25)
=> 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 25th processor, yielding 3.

It is legal to compose ***setf** with **row-major-aref!!**. For example,

```
(*setf (row-major-aref!!
         (!! #2A((0 1) (2 3))) (!! 2)) (!! 25))
```

stores the value 25 in the third element of the component array in each processor.

row-major-sideways-aref!!

row-major-sideways-aref!! *array-pvar row-major-index-pvar* [*Function*]

References the specified array pvar as if it were a vector pvar, with indices taken in row-major order. The result is returned as a pvar.

The *array-pvar* argument must be a slicewise (sideways) array pvar. See the reference entry for ***sideways-array** on page 33 of the *Supplement to the *Lisp Reference Manual* version 5.0.

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.

For example,

```
(*proclaim '(type (array-pvar single-float '(2 2)) foo))
(*defvar foo (!! #2A((5 8) (3 0))))


(defun example ()
  (*slicewise foo)
  (pref (row-major-sideways-aref!! foo (!! 2)) 0)
  )
(example) ==> 3
```

In each processor, is stored the array

5 8
3 0

The element with row-major index 2 is referenced using **row-major-sideways-aref!!**. This results in a pvar whose value is 3 everywhere. The **pref** function then references this value in the 25th processor, yielding 3.

It is legal to compose ***setf** with **row-major-sideways-aref!!**. For example,

```
(*setf (row-major-sideways-aref!! foo (!! 2)) (!! 25))
```

stores the value 25 in the third element of the component array in each processor.

set-vp-set-geometry

set-vp-set-geometry *vp-set geometry-id* [*Function*]

Assigns the specified geometry to the specified VP set.

The parameter *vp-set* must be an instantiated VP set.

The parameter *geometry-id* must be a geometry object created with **create-geometry**. The number of VP's specified by the geometry must be the same as the total number of VP's in *vp-set*.

This function changes the grid shape of CM data. To use this operation properly, it is necessary to understand the mapping of VP's to physical CM processors. See the *Paris Reference Manual* version 5.0, Chapter 2.

Example code:

```
(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)
```

See page 23 of these release notes for information on preventing a geometry from being deallocated when **deallocate-vp-set** is called.

---

sideways-array-p

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

Tests the specified array pvar, returning **t** if it is sideways (slicewise) and **nil** otherwise.

For information on giving an array pvar a sideways orientation, see ***sideways-array** on page 33 of the *Supplement to the *Lisp Reference Manual* version 5.0, and see the documentation for ***slicewise** in these release notes, below.

---

*slicewise

***slicewise** *array-pvar* [**Defun*]

Converts a normal, processorwise array to the slicewise (sideways) orientation.

The *array-pvar* parameter must be a normal, processorwise array. If *array-pvar* is already slicewise (sideways), an error is signaled.

For restrictions on arrays that can be turned sideways (slicewise), see ***sideways-array** on page 33 of the *Supplement to the *Lisp Reference Manual* version 5.0.

Note that ***slicewise** is equivalent to ***sideways-array** when ***sideways-array** is called on an array with a normal, processorwise orientation.

---

*trace

***trace &rest** **defun-function-names* [*Macro*]

***untrace &rest** **defun-function-names* [*Macro*]

Enable and disable tracing for the named parallel functions, which must have been defined using ***defun**.

These macros are similar to the Common Lisp **trace** and **untrace** functions, defined in *Common Lisp: The Language*.

Invoked at top level, **(*trace foo)** causes a message to be printed whenever the function **foo** is either called or exited; **(*untrace foo)** turns off this tracing mechanism.

---

un*defun

un*defun &rest **defun-names* [*Function*]

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 names for functions that have previously been defined with ***defun**. Any number of names may be provided.

When we call **(*defun foo ...)** a macro named **foo** is created. Also, another function with a name derived from **foo** is created. If we subsequently call **(un*defun foo)**, the macro binding is removed from **foo** and the function binding is removed from the symbol with the derived name.

---

vector!!

vector!! &rest *pvars* [*Function*]

Creates and returns a vector pvar containing the specified pvars.

The function **vector!!** takes any number of pvar expressions and returns a vector pvar equal in length to the number of expressions supplied. Each element of the returned vector pvar is a vector that contains one element of each supplied pvar.

For example,

```
(pref (vector!! (self-address!!) (self-address!!)) 25)
=> #(25 25).
```

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.)

The **vector!!** function is similar to the **typed-vector!!** function. However an element-type argument is not required for **vector!!**.

---

vp-set-deallocated-p

vp-set-deallocated-p *vp-set* [*Function*]

Determines whether the specified VP set has been deallocated by a call to **deallocate-vp-set**. (See *Supplement to the *Lisp Reference Manual* version 5.0, page 66.)

This function returns **t** if a VP set has been deallocated, and **nil** otherwise.

---

vp-set-rank
vp-set-total-size
vp-set-vp-ratio

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

vp-set-total-size *vp-set* [*Function*]

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

These functions return the number of dimensions, the total number of processors, and the VP ratio, respectively, for the specified VP set.

---

with-processors-allocated-for-vp-set

with-processors-allocated-for-vp-set *vp-set* [*Macro*]
&key :dimensions :geometry
&body *body*

This macro expands into a form that executes **allocate-processors-for-vp-set** using the specified dimensions and geometry as arguments. It then executes the body of the macro. Finally, **deallocate-processors-for-vp-set** is called to complete the form.

3.2 *Lisp Language Enhancements

3.2.1 Overview of Language Enhancements

The following categories of *Lisp features have enhanced capabilities that distinguish version 5.1 of the *Lisp language from version 5.0:

- *Deallocating VP Sets.* A new optional argument to **deallocate-vp-set** supports deallocating a VP set without deallocating the geometry currently associated with it.

- **Cold-boot.* A new keyword option to the ***cold-boot** operation makes it possible to deallocate all *defvars and all VP sets while cold booting.

- *Scan Operations.* A number of improvements to the scanning features are included in 5.1.

  The **rank!!** and **sort!!** operations can now use segment pvars and can now be done in NEWS order.

  The **scan!!** operation is now considerably faster when the scan function is ***!!** and the pvar contains floating-point data. Also, **scan!!** has a new keyword argument, **:identity**, which must be used if a non-standard scan operator is provided.

  Segment set accessors are now available in the *Lisp package. These extract values from segment set structures, which are created with **create-segment-set** and which are used with the experimental **segment-set-scan!!** function.

- *Type Casting.* It is now possible to treat a portion of an object of one type as if it were of a different type. The **taken-as!!** function has a new optional argument that adds an offset to the source object.

- *Speed.* Several *Lisp operations now execute more quickly. Notably, ***news** and **dot-product!!** are faster for all cases, and **news!!** is faster when fetching data across a power-of-two grid distance.

3.2.2 Description of Language Enhancements

The following *Lisp operations are enhanced in *Lisp version 5.1:

***cold-boot**
segment set accessors
deallocate-vp-set
rank!!
scan!!
sort!!
taken-as!!
ppp

A description of each change follows.

***cold-boot**

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

The **:undefine-all** keyword argument is new. It defaults to **nil**.

If the **:undefine-all** keyword argument is set to **t**, then an invocation of ***cold-boot** deallocates and destroys all pvars and all VP sets—with the exception of the default VP set and the default geometry.

See page 78 of the *Supplement to the *Lisp Reference Manual* version 5.0 for a discussion of ***cold-boot**.

Segment Set Accessors

The parallel and front-end segment set accessor functions were unintentionally omitted from the *Lisp package in version 5.0. They are included in the 5.1 *Lisp package.

segment-set-start-bits!! *segment-set-pvar* [*Function*]

segment-set-start-bits *segment-set* [*Function*]

segment-set-end-bits!! *segment-set-pvar* [*Function*]

segment-set-end-bits *segment-set* [*Function*]

segment-set-processor-not-in-any-segment!! *segment-set-pvar* [*Function*]

segment-set-processor-not-in-any-segment *segment-set* [*Function*]

segment-set-start-address!! *segment-set-pvar* [*Function*]

segment-set-start-address *segment-set* [*Function*]

segment-set-end-address!! *segment-set-pvar* [*Function*]

segment-set-end-address *segment-set* [*Function*]

Each segment set accessor function returns the value of a specific slot in the specified segment set structure. Both parallel and front-end version are provided.

The parallel version of each accessor function takes a segment set pvar. A segment set pvar is a structure pvar returned by a call to **create-segment-set!!**. The parallel accessor functions each return a pvar that contains the parallel values of a single slot in the segment set structure pvar.

The front-end version of each accessor function takes a segment set, which is a front-end structure that corresponds to the segment set structure pvar.The front-end accessor functions each return the value of a single slot of a front-end segment set structure.

For information about the components of a segment set structure pvar, see Chapter 9, "Scanning with Segment Sets," in the *Supplement to the *Lisp Reference Manual* version 5.0. See Chapter 4, "Structure Pvars," in the *Supplement to the *Lisp Reference*

Manual version 5.0 for more information about the relationship between the parallel and front-end structures created by ***defstruct**.

---

deallocate-vp-set

deallocate-vp-set *vp-set* **&optional** *deallocate-geometry-p* [*Function*]

The optional argument, **deallocate-geometry-p** is new. It takes a boolean value, which determines whether the current geometry of the specified VP set is to be deallocated. The default is **t**; the current geometry is deallocated by default.

Usage Note: Any **let-vp-set** form automatically calls **deallocate-vp-set** using the default. Do not assign a geometry that should be preserved to a temporary VP set created with **let-vp-set**.

---

rank!!

rank!! *pvar predicate* **&key :dimension :segment-pvar** [*Function*]

It is now possible to obtain a ranking along grid dimensions and within segments. The keywords, **:dimension** and **:segment-pvar**, now serve the same purpose for **rank!!** as they do for **scan!!**.

The **:dimension** keyword specifies whether the ranking is done separately for each value of a single dimension and, if so, for which dimension. It defaults to **nil**, specifying a send-order ranking. If a value is supplied, it must be an integer, 0 or greater and less than ***number-of-dimensions***. If a dimension is specified, then the ranking is done independently for each row of that dimension.

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, **t** in the first processor of each segment and **nil** in all other processors. (See the **Lisp Reference Manual* version 5.0, page 46, for a further discussion of segment pvars.) If a segment pvar is specified, then the ranking is done independently within each segment.

If both a dimension and a segment pvar are specified, then the ranking is done independently within each row of the dimension and independently within segments within each row.

Example:

Using a **:segment-pvar** argument, we might write:

```
(rank!! (random!! (!! 10))
        '<=!! :segment-pvar
         (evenp!! (self-address!!)))
```

If the first 12 random elements were

0 2 4 2 1 7 5 3 4 7 8 2

then the result would be

0 1 1 0 0 1 1 0 0 1 1 0

Suppose we had a (very small) CM configured with 16 processors in a 4×4 VP geometry, and a pvar *x*, whose values were

1.2	3.4	0.6	-2.3
2.3	-9.3	2.1	-1.2
0.2	1.2	-7.2	0.0
-4.5	3.8	8.1	0.1

Using a **:dimension** argument, we might write:

```
(rank!! X '<=!! :dimension 1)
```

and the result would be

2	2	1	0
3	0	2	1
1	1	0	2
0	3	3	3

scan!!

> ** ** Experimental ** **
>
> *The* **scan!!** **:identity** *feature and the generalized scan concept are experimental. *Lisp programmers are welcome to try them at their own risk and with the understanding that, if insufficiently popular, it may not be supported in future releases.*

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

Previously, only specialized scans were supported in *Lisp. Now, the new **:identity** keyword argument allows generalized scans. This means that parallel scanning calculations are no longer restricted to built-in *Lisp functions. Any associative binary function may now be supplied as the value of *function*. It is an error if the function is not associative.

The familiar, specialized scan uses one of the following predefined parallel functions:

+!!	***!!**		
and!!	**or!!**	**min!!**	**max!!**
logand!!	**logior!!**	**logxor!!**	
copy!!			

For specialized scans, it is an error to specify the **:identity** keyword.

If a user-defined function is specified, a general scan is performed and an **:identity** keyword value must be supplied.

If supplied, the value of **:identity** must be the parallel identity element for *function*. That is, if *function* is applied to the identity pvar in combination with any legal *pvar* value *V*, then the result is *V*.

For instance, a function that does $2x2$ parallel matrix multiplication could be given as the value of *function*.

```
(scan!! my-parallel-matrix 'my-matmult2x2!!
        :identity (!! (make-array '(2 2)
                                  :initial-contents '((1 0) (0 1)))))
```

This specifies the identity matrix as the identity element for the matrix multiply scan.

Performance Notes: Providing a general function to **scan!!** results in significantly slower performance than providing one of the standard, specialized functions.

Previously, if the value of the *pvar* argument was a floating-point pvar and *function* was *!!, execution was slow. This is now remedied.

---

sort!!

sort!! *pvar predicate* [*Function*]
&key :dimension :segment-pvar :key

In all active processors, rearranges the specified pvar data. Data is sorted into ascending order.

The *pvar* argument must be either an integer pvar or a floating-point pvar.

The *predicate* argument must be the symbol **'<=!!**.

The keywords, **:dimension** and **:segment-pvar** are new with version 5.1. They serve the same purpose for **sort!!** as they do for **scan!!**. The keyword **:key** is also new.

The **:dimension** keyword specifies whether sorting is done separately for each value of a single dimension and, if so, for which dimension. It defaults to **nil**, specifying a send order sort. If a value is supplied, it must be an integer, 0 or greater and less than ***number-of-dimensions***. If a dimension is specified, then sorting is done independently for each row of that dimension.

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. (See the **Lisp Reference Manual* version 5.0, page 46, for a further discussion of segment pvars.) If a segment pvar is specified, then sorting is done independently within each segment.

If both a dimension and a segment pvar are specified, then sorting is done independently within each row of the dimension and independently within segments within each row.

The **:key** argument provides the key on which the sort is done. If provided, it must be a function that takes one pvar argument and returns a pvar. The **:key** function is called

on the supplied pvar and then the sort comparisons are done on the value of the result in each active processor.

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!!** of this description would rearrange data based on the value of the accessed slot in each processor.

Examples:

Let * represent an unselected processor. Assume we have an 8 processor CM and a pvar with the following values:

7 * 2 3 * 1 0 6

The result of calling **sort!!** on this pvar is

0 * 1 2 * 3 6 7

Notice that data in unselected processors remains unchanged.

Using a **:segment-pvar** argument, we might write:

```
(sort!! (random!! (!! 10))
        '<=!! :segment-pvar (evenp!! (self-address!!)))
```

If the first 12 random elements were

0 2 4 2 1 7 5 3 4 7 8 2

then the result would be

0 2 2 4 1 7 3 5 4 7 2 8

Using a **:key** argument, we might write

```
(sort!! afoo '<=!! :dimension 0 :key 'foo-a!!)
```

If **afoo** is an instance of a parallel structure with slot **foo-a!!**, then this form sorts **afoo** using its **a** slot as the key. The sort occurs independently along each row of dimension 0 (the *x* dimension).

taken-as!!

taken-as!! *pvar pvar-type* **&optional offset** [*Function*]

The new optional argument, **offset**, determines whether the pvar data to be taken as the specified type should first be offset and, if so, by how much. The default is 0; no offset is taken by default.

Example:

Consider the pvar **u16**, which is 16 bits long.

```
(*proclaim '(type (pvar (unsigned-byte 16)) u16))
(*defvar u16)


(need-only-8-bits
  (taken-as!! u16 '(pvar (unsigned-byte 8)) 4)
  )
```

The function **need-only-8** requires an 8-bit pvar. Using **taken-as!!** on **u16** with an offset argument of 4 causes the 4th through the 11th bit of **u16** in each processor to be manipulated by **need-only-8** as an **(unsigned-byte 8)** pvar.

---

ppp

ppp *pvar* [*Macro*]

The *Lisp pretty-print macro, **ppp**, is enhanced. Structures are now formatted in a much more readable manner and two new keywords are supported: **:pretty** and **:stream**.

In 5.1, calling **ppp** on a structure, by default, 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 new keyword **:pretty** is given the value **t**, printed output such as the following results:

```
#S(PERSON :NAME 0
      :AGE 0
      :SEX NIL)
#S(PERSON :NAME 0
      :AGE 0
      :SEX NIL)
#S(PERSON :NAME 0
      :AGE 0
      :SEX NIL)
```

The new **:stream** keyword takes a stream and directs output there. The default is standard output.

4 *Lisp Compiler Version 5.1

The *Lisp compiler is an extension to the Common Lisp compiler as implemented in each Connection Machine front-end development environment. The *Lisp compiler translates *Lisp code into Common Lisp code with calls to Paris. Then the installed Common Lisp compiler translates the Lisp code generated by the *Lisp compiler into native machine instructions.

Version 5.1 of the *Lisp compiler handles an expanded set of *Lisp operations and generates more efficient code for many of the operations it compiles.

NOTE: Here, as in other *Lisp documentation, the verb "to *compile" is used to mean "to compile with the *Lisp compiler." In this way, compilation by the Common Lisp compiler (*x* is compiled) is distinguished from compilation by the *Lisp compiler (*x* is *compiled).

4.1 *Lisp Compiler Enhancements

Several improvements distinguish version 5.1 of the *Lisp compiler from version 5.0.

- *Increased Scope.* More *Lisp operations now *compile. The new ones are documented below.
- *Improved Performance.* The *Lisp compiler produces more efficient code than previously.
- *New and Enhanced Compiler Options.* Several new options are supported and one existing option is improved.

4.1.1 Increased Scope—Forms That Newly *Compile

Not all but most *Lisp operations can be *compiled; those that cannot be *compiled run interpreted. For a general discussion about what does get *compiled, see the **Lisp Compiler Guide* version 5.0.

Most operations new with version 5.1 can be *compiled.

4.1.1.1 Forms That *Compile without Restrictions

In version 5.1, the *Lisp operations listed below *compile. In previous versions, these were not handled by the *Lisp compiler.

(!! *CL-aggregate*)
allocate!!
amap!!
array-in-bounds-p!!
array-row-major-index!!
byte!!
***defvar**
dot-product!!
news-border!!
pref
ppp!!
(*setf (pref *parallel-structure*))
sideways-aref!!
typed-vector!!
v+!!
v-!!
v*!!

In addition, the *Lisp compiler attends to the special forms ***let**, ***let***, **let**, **let***, **compiler-let**, and **progn** when they appear inside *Lisp forms that *compile.

Descriptions follow as warranted.

Parallel Aggregates *Compile.

In version 5.1, the *Lisp compiler compiles aggregate parallel data. Simple parallel data constructs of the form (!! *x*) have always *compiled—if properly declared. Now parallel vectors, parallel arrays, and parallel structures also *compile.

For example, in addition to expressions such as

```
(*set (the (unsigned-byte-pvar 8) x) (!! 5))
```

each of the following expressions also *compiles:

```
(*set (the (vector-pvar single-float 3) x)
       (!! #(1.0 2.0 3.0)))

(*set (the (array-pvar (unsigned-byte 8) (2 2)) x)
      (!! #2A((0 1) (2 3))))

(*set (the foo-pvar afoo) (!! #S(FOO :A nil :B 0.0)))
```

Some Parallel Vector Functions *Compile.

The parallel vector addition, subtraction, and multiplication functions, v+!!, v-!!, and v*!! all *compile. For example,

```
(*proclaim '(type (vector-pvar single-float 3) x y z))
(*defvar x)
(*defvar y)
(*defvar z)

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

now *compile. The equivalent form, using **amap!!**, also *compiles:

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

Parallel vector functions compile into very efficient code.

For instance,

```
(*set v1 (v+!! v2 (v*!! v3 v4)))
```

where the v's are vector pvars, *compiles into extremely efficient code.

The **dot-product!!** function *compiles.

For instance, the following expression *compiles:

```
(*set (the single-float-pvar q) (dot-product!! x))
```

Notice that this can be rewritten using **reduce!!** and **amap!!** as

```
(*set (the single-float-pvar q)
      (reduce!! '+!! (amap!! '*!! x x)))
```

which also *compiles.

Some Special Forms *Compile.

The special forms listed below now *compile when they occur inside any *Lisp form that *Lisp compiler compiles. (See *The *Lisp Compiler Guide* version 5.0, page 5, for a list of *Lisp macros that compile.)

***let**	***let***
let	**let***
compiler-let	**progn**

For example:

```
(*set sf3 (*let ((sppoo (sqrt!! sf2)))
            (declare (type single-float-pvar sppoo))
            (sqrt!! sppoo)))
```

As a further example, consider the the following expression containing nested **compiler-let** forms:

```
(*set sf1 (compiler-let ((*safety* 0))
            (/!! sf1 (compiler-let ((*safety* 3))
                       (+!! sf2 sf3)))))
```

The +!! is compiled with safety 3, the /!! with safety 0, and the ***set** with whatever it was before this call.

With the ability to *compile these special forms comes the ability to use Paris-like functions with negligible overhead, as in the code below.

```
(defmacro *lisp::my-function (argument)
  `(*let (destination)
     (declare (type (pvar single-float) destination))
     (cmi::my-function (pvar-location destination)
                       (pvar-location ,argument) 23 8)
     destination))

(*set dest (*lisp::my-function source))
```

Usage Note: Generally, only the initial value forms of a ***let** or ***let*** are *compiled. However, if a ***let** or ***let*** occurs within another form that the *Lisp compiler compiles, then the body of the ***let** or ***let*** is also *compiled.**

4.1.1.2 Forms That *Compile with Restrictions

In version 5.1, the following *Lisp operations, which did not previously *compile now *compile if the restrictions noted below are observed:

dpb!!	**ldb!!**
ldb-test!!	**mask-field!!**
pref!!	**scan!!**

A few *Lisp operations that *compile with restrictions were poorly documented in the 5.0 release notes. For the operations listed below, more accurate descriptions of applicable compiler restrictions are included here.

character!!	**code-char!!**
digit-char-p!!	**int-char!!**
make-char!!	
***let**	***let***
***unless**	***when**

---

dpb!! **ldb!!**
ldb-test!! **mask-field!!**

The *bytespec-pvar* argument must textually be a call to **byte!!**.

---

pref!!

The argument *pvar-expression* must be a simple expression, such as a variable.

A previously undocumented restriction on the **pref!!** macro has been lifted in 5.1. The **pref!!** macro now *compiles when the **:collision-mode** keyword value is **:many-collisions**. In the past this has not been the case.

---

scan!!

Previously, the **scan!!** function *compiled. However, **scan!!** did not accept ***!!** as its *function* argument. With version 5.1, ***!!** may be specified as the *function* argument. In order to *compile a **scan!!** with a *function* argument of '***!!**, the *pvar* argument must be a floating-point pvar.

---

character!! **code-char!!**
digit-char-p!! **int-char!!**
make-char!!

These *compile only if the result is used as the source pvar in a ***set** form.

***let**
let

Only the initial values are guaranteed to *compile. The *Lisp compiler attends to the body as to any other form; operations that otherwise *compile do so inside the body of a ***let** or ***let***. Operations that do not otherwise *compile do not *compile from within one of these forms.

***unless**
***when**

Only the predicate is guaranteed to *compile. The *Lisp compiler attends to the body as to any other form; operations that otherwise *compile do so inside the body of a ***when** or ***unless**. Operations that do not otherwise *compile do not *compile from within one of these forms.

4.1.2 Improved Performance

The following categories of *Lisp operations now compile more efficiently:

- Combinations of parallel multiplication and addition
- Long distance NEWS communication
- Context selection forms

Descriptions follow.

For Connection Machines with floating-point hardware, combinations of *!! and +!! or of v*!! and v+!! now produce very efficient compiled code.

An example of the affected expressions is:

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

where *x*, *y*, and *z* are floating-point pvars.

Long-distance **news!!** communication is faster.

If the **news!!** distance is a power of two, **news!!** compiles into one of the new Paris **CM:get-from-power-two** instructions, and is therefore faster.

In version 5.1, the context-selection forms, ***all**, ***when**, ***let**, and ***defun**, produce more efficient expansions.

The compiler now discerns when one of these forms is returning a pvar and handles it efficiently, even when several of these forms are nested.

Usage Note: If the last form of a ***all** or a ***when** is a ***all** or a ***when**, then the inner form does not save and restore the context flag.

4.1.3 New and Enhanced Compiler Options

Version 5.1 includes a few changes to the compiler options. The **Use Always Instruction** compiler option is improved and a new option, **Rewrite Arithmetic Expressions**, is supported for *Lisp environments on any front end.

Three options have been added to the *Lisp compiler menu on Symbolics front ends.

- Macroexpand Repeat
- Macroexpand Inline Forms
- Macroexpand Print Case

The defaults for these three options (**t**, **t**, and **nil**, respectively) correspond to the default behavior of a Symbolics Lisp machine.

Use Always Instruction

This option is imported with version 5.1. The *Lisp compiler now generates Paris **-always** instructions for code inside of a ***all** form when the variable ***use-always-instructions*** is set to **t**.

Macroexpand Repeat

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

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

A value of **t** causes the command **Macro Expand Expression** to use the Common Lisp **macroexpand** function, which macroexpands using **macroexpand-1** repeatedly.

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

Macroexpand Inline Forms

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.

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

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 output of the *Lisp compiler hard 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 the following code:

```
(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 the following code:

```
(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

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-print-case*** value of **nil** causes the value of the variable ***print-case*** to be used. This is the default.

If the value of ***macroexpand-print-case*** is non-nil, it is used.

Rewrite Arithmetic Expressions

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

This option determines whether the compiler optimizes arithmetic operations as if they were associative.

A value of of **t** allows the compiler to rewrite arithmetic operations as if they were associative. This is the default.

A value of **nil** prevents this arithmetic-rewriting optimization.

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

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

The laws or 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.

When this option is enabled (the default), the *Lisp compiler may produce more efficient code.

When this option is disabled, the *Lisp compiler evaluates expressions in the order in which they appear textually (the second alternative above).

Regardless of the value of ***rewrite-arithmetic-expressions***, the user may force a specific order of evaluation by explicitly directing the computation, as in the following:

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

4.2 *Lisp Compiler Restrictions

4.2.1 New 5.1 Forms That Don't *Compile

The operations listed below, which were introduced with version 5.1, do not yet *compile.

deallocate-processors-for-vp-set
deallocate-vp-set-processors
news-direction!!
***news-direction**
off-grid-border-relative-direction-p!!
***processorwise**
***room**
sideways-array-p
***slicewise**
***trace**
***untrace**
un*defun
vp-set-deallocated-p
vp-set-rank
vp-set-total-size

vp-set-vp-ratio
with-processors-allocated-for-vp-set

4.2.2 Cumulative List of Forms That Don't *Compile

For easy reference, here is a cumulative list of all the *Lisp forms that do not compile. (This list supersedes section 5.2.1 of the **Lisp Release Notes* version 5.0.)

address-nth!! **address-plus!!**
address-plus-nth!! **address-rank!!**
array-dimension!! **array-dimensions!!**
***array-dimensions** ***array-element-type**
array-in-bounds-p!! **array-rank!!**
***array-rank** ***array-total-size**
array-total-size!! **array-to-pvar**
array-to-pvar-grid **byte!!**
byte-size!! **byte-position!!**
char-bit!! **deallocate-processors-for-vp-set**
deallocate-vp-set-processors **deposit-field!!**
digit-char!! **equalp!!**
grid!! **grid-relative!!**
***map** ***news**
news-direction!! ***news-direction**
off-grid-border-p!! **off-vp-grid-border-p!!**
off-grid-border-relative-direction-p!!
***processorwise** **pvar-to-array**
pvar-to-array-grid ***pset-grid-relative**
ppp-address-object **pppdbg**
***room** **rot!!**
set-char-bit!! **sideways-array-p**
***slicewise** **sort!!**
***sideways-array** **structurep!!**
***trace** ***untrace**
un*defun **vp-set-deallocated-p**
vp-set-rank **vp-set-total-size**
vp-set-vp-ratio
with-processors-allocated-for-vp-set
parallel-structure-**p!!**

In addition, none of the experimental parallel sequence operations can be *compiled, nor can any of the experimental segment set scan operations.

5 *Lisp Simulator Version 5.1

The *Lisp simulator runs on top of Common Lisp and executes *Lisp code without using a Connection Machine system.

The *Lisp simulator is known to run on the following implementations of Common Lisp:

Symbolics Common Lisp on a Symbolics Lisp machine
Many Common Lisp implementations on many machines running UNIX
Allegro Common Lisp on a Macintosh II

Thinking Machines Corporation customer support can provide tapes and installation instructions.

The *Lisp simulator can be made to run on any full implementation of Common Lisp with minimal porting effort.

5.1 New Simulator Version

Version F16 of the *Lisp simulator corresponds to *Lisp version 5.1.

Version F15 of the *Lisp simulator, which corresponds to *Lisp version 5.0, was shipped with *Lisp version 5.1. Version F16 of the *Lisp simulator is now available for general release. If you wish to have F16 installed, ask your applications engineer or a Thinking Machines Corporation customer support representative to do so.

5.2 *Lisp Simulator Enhancements

5.2.1 Version 5.1 Language Features Simulated

All the language features that are either new or enhanced in *Lisp version 5.1 are implemented in the *Lisp simulator.

5.3 Simulator Restrictions

The *Lisp simulator supports only general pvars; it does not support any of the other pvar data types (e.g., floating-point pvars or complex pvars). The *Lisp simulator does, however, support aggregate data structures, such as array pvars, vector pvars, and structure pvars.

5.3.1 Restrictions Lifted in 5.1

With the exception of the restriction on pvar type, all the simulator restrictions reported in section 6.2 of the **Lisp Release Notes* version 5.0 are lifted with the release of version 5.1.

The patch to ***pset**, given in section 6.3 of the **Lisp Release Notes* version 5.0, is incorporated in versions F15 and F16 of the *Lisp simulator.

5.3.2 Abort and Cold Boot Problem

If the *Lisp simulator is aborted in the wrong place, an attempted ***cold-boot** operation will not succeed; the simulator will go into the debugger and not complete. To reset, execute the following forms:

```
(*sim-i::reset-everything)
(*cold-boot)
```

This will generally clear up the problem, albeit at the expense of destroying all *defvar and VP set definitions.

6 Implementation Notes

Information about version 5.1 implementation corrections and about new implementation errors in version 5.1 will be published in successive issues of the system software bulletin, *In Parallel,* beginning in August 1989.

7 Helpful Hint: *set Restriction

A previously undocumented restriction on ***set** has caused errors in some users' code.

It is an error to attempt to ***set** a function parameter if that parameter is a temporary pvar. A temporary pvar is defined as **nil!!**, **t!!**, or any other pvar not created with ***let**, ***let***, ***defvar**, or **allocate!!**. Temporary pvars are created by *Lisp functions such as !! and +!!.

Consider the function **foo** below.

```
(defun foo (x) (*set x (!! 5)))
```

The following expressions violate this rule and are therefore in error:

```
(foo (!! 3))

(foo (cos!! (+!! a b)))
```

However, the expression below is not in error because **x** is not a temporary pvar; **x** is created with ***let**.

```
(foo (*let (x) (*set x (cos!! (+!! a b))) x))
```

The result of violating the restriction against using ***set** to change the value of a temporary variable is undefined. The *Lisp simulator catches the error and prints an error message. Neither the *Lisp interpreter nor the *Lisp compiler catches this error.