The
Connection Machine
System

*Lisp Release Notes

Version 6.0
November 1990

These release notes
replace all previous
release notes

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, November 1990

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

Contents

About These Release Notes

Objectives

The **Lisp Release Notes* Version 6.0 are published to inform *Lisp programmers about all new and changed *Lisp features introduced with the Connection Machine System Software Version 6.0.

Intended Audience

The reader is assumed to have a working knowledge of Common Lisp, as described in *Common Lisp: The Language*, and of *Lisp, as described in the current *Lisp documentation. The reader is also assumed to have a general understanding of the Connection Machine system.

Revision Information

These release notes are new with *Lisp Version 6.0, and *replace all previous release notes.*

Organization of These Release Notes

Related Manuals

- **Lisp Dictionary*

  This manual provides a complete dictionary-format listing of the functions, macros, and global variables available in the *Lisp language. It also includes helpful reference material in the form of a glossary of *Lisp terms and a guide to using type declarations in *Lisp. Except as noted in these release notes, the *Dictionary* is the most accurate and current description of the *Lisp language.

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

  These manuals together provide a conceptual overview of the basic features of the *Lisp language as of Version 5.0. For detailed descriptions of operations they have been superseded by the **Lisp Dictionary.*

- **Lisp Compiler Guide*

  This manual describes the *Lisp compiler.

Note: The **Lisp Reference Manual, Reference Supplement,* and *Compiler Guide* are bound together in a volume entitled *Programming in *Lisp.*

- *Connection Machine Parallel Instruction Set*

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

- *CM User's Guide*

  This document, new with Version 6.0, provides helpful information for users of the Connection Machine system, and includes a chapter devoted to the use of *Lisp and Lisp/Paris on the Connection Machine.

- *Common Lisp: The Language,* Second Edition, by Guy L. Steele Jr. Burlington, Mass.: Digital Press, 1990.

  The first edition of this book (1984) was the original definition of the Common Lisp language, which became the de facto industry standard for Lisp. ANSI technical committee X3J13 has been working for several years to produce an ANSI standard for Common Lisp. The second edition of *Common Lisp: The Language* contains the entire text of the first edition, augmented by extensive commentary on the changes and extensions recommended by X3J13 as of October 1989.

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.

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
Usenet 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.

1 About Version 6.0

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.

Version 6.0 is a major *Lisp release, and includes a number of language enhancements such as new options for several functions and compilation of virtually all operators in the *Lisp language. *Lisp Version 6.0 also provides faster execution of *Lisp code and corrects a number of implementation errors.

An important new feature of the *Lisp language is the elimination of the need to use the **!!** (bang-bang) operator. With few exceptions, functions and macros that accept pvar arguments will now accept scalar arguments as well, and will promote these scalars to constant pvars automatically.

1.1 Components of *Lisp, Version 6.0

Thinking Machines Corporation's implementation of *Lisp includes:

- The *Lisp interpreter, which executes *Lisp code interpretively on the Connection Machine system.
- The *Lisp compiler, which translates *Lisp code into compiled Lisp/Paris code for faster execution.
- The *Lisp simulator, which executes *Lisp code on a serial front-end computer alone, simulating the operations of the Connection Machine system.

The *Lisp interpreter and compiler can be used from within a Common Lisp environment on any Connection Machine (CM) front end. CM front ends currently supported include the Symbolics 3600-series Lisp machine, Sun-4 Workstations running the UNIX operating system, and Digital Equipment Corporation VAX machines running the ULTRIX operating system. The *Lisp simulator can be run on any machine with a Common Lisp language environment. CM hardware is not required to run the simulator.

1.2 *Lisp Documentation

The currently available documentation for *Lisp is listed below.

The following documents provide important conceptual and reference information on the *Lisp language:

- The **Lisp Reference Manual*, Version 5.0, revised October 1988
- The *Supplement to the *Lisp Reference Manual,* Version 5.0, October 1988
- The **Lisp Compiler Guide*, Version 5.0, October 1988
- The **Lisp Dictionary*, Version 5.2, February 1990

The following documents are new as of Version 6.0:

- The *CM User's Guide*, Version 6.0, November 1990
- The **Lisp Release Notes,* Version 6.0, November 1990

The **Lisp Reference Manual* and *Supplement to the *Lisp Reference Manual* together provide important conceptual information on the *Lisp language. However, all reference material in these documents has been superseded by the information contained in the **Lisp Dictionary*. Other than as noted in these release notes, the material in the **Lisp Dictionary* is the most current and correct.

The **Lisp Compiler Guide* provides important information for users of the *Lisp compiler. Users of the compiler will also want to consult the type declaration chapter in the **Lisp Dictionary*, which provides a set of guidelines for properly declaring *Lisp code.

The **Lisp Dictionary* is a complete reference source for *Lisp. It includes a list of all *Lisp operators, descriptions of important global variables, and a complete dictionary entry for each function and macro in the *Lisp package. The **Lisp Dictionary* also includes a glossary of important terms used in *Lisp and a chapter on *Lisp pvar types and type declaration.

The *Connection Machine System User's Guide*, new with Version 6.0, provides helpful information for users of the Connection Machine system, and includes a chapter devoted to the use of *Lisp and Lisp/Paris on the Connection Machine.

Finally, these release notes document all changes to *Lisp as of Version 6.0.

1.3 Summary of Enhancements

The following enhancements distinguish *Lisp Version 6.0 from previous versions.

- **Automatic Promotion of Scalar Arguments to Pvars.** With few exceptions, functions and macros that accept constant pvar arguments will now accept scalar constants as well, and will automatically promote these arguments to constant pvars. For example, the expression

  ```
  (+!! x (!! 2.0) (!! constant))
  ```

  may now be rewritten as

  ```
  (+!! x 2.0 constant)
  ```

- **New Options for *Lisp Operators.** New keyword arguments have been added to the *Lisp operators **scan!!** and ***pset**, a new combiner keyword has been added to ***pset**, and a new global variable has been added to allow user control of the run-time context selection performed by the **pref** operator.

- **New Functions.** The previously internal function **deallocate–geometry** is now available from the *Lisp package, and several new functions have been added.

- **Code Walker Now on by Default.** The code walker portion of the *Lisp compiler is now on by default. It may still be disabled, if necessary; see Section 6.3.1, below.

- **More Complete Compilation of *Lisp Code**. The *Lisp compiler now compiles virtually all the *Lisp language, when proper type declarations are provided.

- **Faster Execution of *Lisp Code**. *Lisp code will in many cases execute faster, because of improved code generation and many improvements in performance at the Paris level. For details of these improvements, see the chapter on Paris in the *CMSS Summary* for Version 6.0.

- **New Version of the *Lisp Simulator.** A new version of the *Lisp simulator, F18, is available. This version of the simulator is fully compatible with Version 6.0 of *Lisp.

- ***Lisp Simulator Now Freely Available.** As of Version 6.0, the *Lisp simulator is freely available for copying and modification by users. For details on where and how to obtain a copy of the simulator, see Section 7.2 of this document.

- **Additions to *Lisp Library.** The *Lisp Library now includes a set of functions for fast data transfer between a front-end disk and the CM.

1.4 *Lisp Software Requirements

*Lisp Version 6.0 requires Lucid Common Lisp Version 2.5 to run on VAX front ends. This is unchanged from the requirements of *Lisp Version 5.2. (See Section 3.1 for more information about Lucid 2.5.)

*Lisp Version 6.0 requires Lucid Common Lisp Version 3.0 to run on Sun-4 front ends. This is unchanged from the requirements of *Lisp Version 5.2. (See Section 3.2 for more information about Lucid 3.0.)

*Lisp Version 6.0 is supported on Symbolics 3600-series front ends under both Genera 7.2 and Genera 8.0.

1.5 *Lisp On-line Code Examples

Examples of *Lisp code are available on-line in the following directories:

```
/cm/starlisp/interpreter/f6000/*example*.lisp
/cm/starlisp/graphics/f6001/examples.lisp
```

Ask your systems administrator or applications engineer to help you locate these files at your site.

Code examples are also available to Connection Machine Network Server users in the CMNS archives directory.

1.6 Miscellaneous New Features

1.6.1 The cm:time Macro Now Nests

As a side effect of the new Paris timing mechanism provided in Version 6.0, calls to the timing macro **cm:time** may now be nested. Each call to **cm:time** allocates a separate timer that is guaranteed to be independent of any timers the user may have otherwise allocated. Prior to Version 6.0, calls to **cm:time** could not be nested.

For more information on the uses of **cm:time**, and for examples of nested calls to **cm:time**, refer to the *CM User's Guide*.

2 Porting Code to Version 6.0

*Lisp Version 6.0 requires Connection Machine System Software Version 6.0.

*Lisp source code written in previous versions of *Lisp (5.0, 5.1, and 5.2) will run under Version 6.0 unchanged. However, *Lisp programs compiled under previous versions must be recompiled to run under Version 6.0. That is, *Lisp programs compiled under versions V5.0, V5.1, and V5.2 must be recompiled to run in Version 6.0.

2.1 Obsolete Language Features

An obsolete language feature is one that is no longer supported and should not be used in *Lisp code. Features documented as obsolete are not guaranteed to work in future versions of the *Lisp language.

Operators reported obsolete prior to Version 6.0 are listed below, along with the operations that should be used in their place.

Obsolete Operator(s):	Replaced by:
dsf–v+!!, dsf–v–!!, dsf–v*!!	**v+!!, v–!!, v*!!**
sf–v+!!, sf–v–!!, sf–v*!!	**v+!!, v–!!, v*!!**
sf–dot–product!!	**dot–product!!**
pref–grid	**pref** with **grid**
pref–grid!!	**pref!!** with **grid!!**
pref–grid–relative!!	**news!!, pref!!**
***pset–grid**	***pset** with **grid!!**
***pset–grid–relative**	***news, *pset**
scan–grid!!	**scan!!** with **:dimension** keyword
(setf (pref ...))	**(*setf (pref ...))**
(setf (pref!! ...))	**(*setf (pref!! ...))** or ***pset**
with–*lisp–from–paris	No longer needed
with–paris–from–*lisp	No longer needed

As of Version 6.0, the following operators are also obsolete:

Obsolete Operator(s):	Replaced by:
dsf–cross–product!!	**cross–product!!**
dsf–vector–normal!!	**vector–normal!!**
dsf–vscale–to–unit–vector!!	**vscale–to–unit–vector!!**
sf–cross–product!!	**cross–product!!**
sf–v+–constant!!	**v+scalar!!**
sf–v—constant!!	**v–scalar!!**
sf–v*–constant!!	**v*scalar!!**
sf–v/–constant!!	**v/scalar!!**
sf–vabs!!	**vabs!!**
sf–vabs–squared!!	**vabs–squared!!**
sf–vector–normal!!	**vector–normal!!**
sf–vscale–to–unit–vector!!	**vscale–to–unit–vector!!**

These functions were introduced to provide optimized interpreter performance for floating-point vector operations. The *Lisp compiler now compiles the corresponding general vector operators. It is recommended that the general vector functions be used rather than the above interpreted operations.

3 *Lisp and Lucid Common Lisp

*Lisp requires different versions of Lucid Common Lisp on VAX and on Sun-4 front ends. These version requirements apply equally to the *Lisp interpreter and compiler. *Lisp programmers are strongly advised to obtain Lucid Common Lisp documentation appropriate to their front-end environment.

*Lisp Version 6.0 on a VAX front end requires Lucid Common Lisp 2.5.
*Lisp Version 6.0 on a Sun-4 front end requires Lucid Common Lisp 3.0.

3.1 Lucid Common Lisp Version 2.5 on VAX Front Ends

The release notes for Lucid Common Lisp Version 2.5 completely detail how Version 2.5 differs from Version 2.1. The Lucid changes between these versions that most affect *Lisp programs are noted below.

(1) **Use of the change–memory–management function discouraged**

The **change–memory–management** function is not recommended by Lucid for Version 2.5. Only users with programs that contain large amounts of code or code that requires heavy garbage collection need to use this function. Ask your site manager, or contact Thinking Machines Corporation customer support, for assistance in determining the proper arguments to supply to this function.

(2) **Foreign function interface changed**

The Lucid foreign function interface differs between Version 2.1 and Version 2.5. Consult the Lucid Common Lisp Version 2.5 documentation for details.

(3) **Lucid ephemeral garbage collector**

In prior Lucid releases, garbage collection occurred frequently and took significant amounts of time. Lucid Common Lisp Version 2.5 includes an ephemeral garbage collector. Consequently, full garbage collection is neither as frequent nor as noticeable.

3.2 Lucid Common Lisp Version 3.0 on Sun-4 Front Ends

Lucid Common Lisp Version 3.0 is significantly different from Lucid 2.1. The Lucid changes that most affect *Lisp programs are noted below.

(1) **Name changes**

The naming convention for certain Lucid functions differs between Version 2.1 and Version 3.0. 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 should not be necessary for most *Lisp applications. Only users with programs that contain large amounts of code or code that requires heavy garbage collection need to use this function. If this function is used, it is recommended that the following form be executed immediately after starting up a *Lisp environment:

```
(lcl:change-memory-management :expand-reserved 50
                              :expand-p t)
```

For extremely large programs, the expansion value of 50 can be replaced by 75 or 100. If running *Lisp code causes excessive garbage collection thereafter, the following form may help:

```
(lcl:change-memory-management :expand 50 :expand-p t)
```

Compilation of large amounts of code can also cause heavy garbage collection. Compiling with the Lucid development compiler rather than the Lucid production compiler reduces the amount of garbage collection and the compilation time, at the expense of losing some front end performance.

The output of the Lisp expression **(room t)** includes information about the current memory management settings.

(3) **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.

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

```
;; Put the Lucid 3.0 compiler in production mode.

(defun prod ()
  (if (find-package '*lisp)
      (starlisp-prod)
      (proclaim '(optimize (compilation-speed 0)
                           (safety 1) (speed 3)))))

(defun starlisp-prod ()
   (eval (read-from-string
           "(funcall *LISP-I::*OLD-PROCLAIM-FUNCTION*
              '(optimize  (compilation-speed 0)
                          (safety 1) (speed 3)))")))


;; Put the Lucid 3.0 compiler in development mode.

(defun dev ()
  (if (find-package '*lisp)
      (starlisp-dev)
      (proclaim '(optimize (compilation-speed 3)
                           (safety 3) (speed 2)))))

(defun starlisp-dev ()
   (eval (read-from-string
           "(funcall *LISP-I::*OLD-PROCLAIM-FUNCTION*
              '(optimize  (compilation-speed 3)
                          (safety 3) (speed 2)))")))
```

The settings used in these functions 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 these function definitions. Using the development compiler can significantly speed up the compilation process.

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

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.

(4) **Foreign function interface changed**

The Lucid foreign function interface differs between Version 2.1 and Version 3.0. Consult the *Lucid 3.0 Advanced User's Guide* for details.

(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.

3.3 Lucid-related Implementation Errors and Restrictions

Most previously reported Lucid-related implementation errors have been corrected for the release of *Lisp Version 6.0. The known outstanding bugs and restrictions are reported again in these release notes. All past issues of Programming in *Lisp *In Parallel* and all previous **Lisp Release Notes* may therefore be discarded.

3.3.1 Known Errors Corrected

The following Lucid-related implementation error reported in *In Parallel* Vol. 3, No. 1, March 1990, is fixed in *Lisp Version 6.0:

implicit–return–pvar–p–hangs

3.3.2 Known Errors Still Open

All known Lucid-related implementation restrictions for Version 6.0 *Lisp are reported here. If new bugs are discovered, they will be reported during the coming months in the *In Parallel* software bulletin, Vol. 4.

ID load–n–defstruct–wrong–warning

Environment

DFS, *Lisp Version 5.1 and 5.2; Sun-4 front end

Description

Performing **load–n** of a file with a **defstruct** form generates a spurious warning message.

Reproduce By

If the file **"/mydir/load-n-warning-bug"** contains the code

```
(in-package '*lisp)
(defstruct pfoo-1
  a
  b)
```

then, in a Lucid Lisp environment on a Sun-4,

```
(dfs:load-n "/mydir/load-n-warning-bug" :recompile t)
```

produces the following messages when compiled:

```
;;; Compiling /mydir/load-n-warning-bug.lisp
;;; Reading source file "/mydir/load-n-warning-bug.lisp"
;;; Writing binary file "/mydir/load-n-warning-bug.sbin3"
;;; Loading binary file /mydir/load-n-warning-bug.sbin3

WARNING:  the definition of PFOO-1 by DEFSTRUCT in the file
#P"/mydir/load-n-warning-bug.lisp"
is being overridden by DEFSTRUCT in the file
"/mydir/load-n-warning-bug.lisp".

WARNING: PFOO-1 is multiply defined in the file
/mydir/load-n-warning-bug.lisp

#P"/tmp_mnt/am/krnl/datascope/load-n-warning-bug.sbin3"
```

Workaround

Ignore the duplicate error message.

Status

Open.

ID lucid–floating–point–compiler–bug

Environment

Lucid Common Lisp, Version 3.0

Description

Uncorrected bug in Lucid production compiler causes lexical scoping problems in code that includes floating-point variables.

Reproduce By

```
(proclaim '(optimize (compilation-speed 0)))
```

```
(defvar global 0.0)
(defvar anything 0.0)

(compile (defun bug ()
   (let ((x 0.0))
      (declare (type single-float x))
      (setq x (let ( (local global)
                     (anything 0.0))
                 (print (prog1 (identity 9.0)
                               (print local)))))
    (format t  "~%VALUE OF X (should be 9.0)=~s~%~s"
               x (if (equal x 9.0) 'OKAY 'BUG))
    (values))))
(bug)
0.0
9.0
VALUE OF X (should be 9.0)=0.0
BUG
```

Workaround

Use the Lucid development compiler :

```
(proclaim '(optimize (compilation-speed 3)))
```

For example, after recompiling the definition of **bug** with the Lucid development compiler, the following output is displayed:

```
(bug)
0.0
9.0
VALUE OF X (should be 9.0)=9.0
OKAY
```

Not declaring the affected variable (**x** in the above example) to be a floating-point value is another workaround.

Status

Open.

ID lucid–byte–specifier–size–limit

Environment

Lucid Common Lisp, Versions 2.5 and 3.0

Description

Lucid Common Lisp imposes a limit on the size of byte-specifier data objects. If either argument to the **byte** operation is greater than 4095, an error is signalled.

Reproduce By

```
> (byte 4095 0)
#.(BYTE 4095. 0.)

> (byte 4096 0)
>>Error: The byte specified for BYTE, [size=4096,
position=0], is not within the range of byte-specifiers.
BYTE:
   Required arg 0 (SIZE): 4096
   Required arg 1 (POSITION): 0
:C  0: Supply new size and position arguments.
:A  1: Abort to Lisp Top Level
```

Workaround

Use the operations **load–byte** and **deposit–byte**, which permit independent specification of byte size and position arguments.

Status

Open.

4 *Lisp Language Version 6.0

This section describes the following changes and additions made to the *Lisp language in Version 6.0:

- Automatic promotion of scalar arguments by *Lisp functions
- New options for several *Lisp operators
- Several new functions added to the *Lisp package

4.1 Automatic Promotion of Scalar Arguments

Previously, in order to supply a constant pvar argument to a *Lisp operator, it was necessary to use the **!!** operator, as in the following function call:

```
(+!! pvar-x (!! 3) (!! constant))
```

As of Version 6.0 of *Lisp, virtually all the functions and macros in *Lisp that accept constant pvars as arguments will now accept scalar constants as well, and will automatically convert those scalars into constant pvars, as if via a call to **!!**. So, for example, the above function call could be rewritten as:

```
(+!! pvar-x 3 constant)
```

This feature is available within function definitions, as well. For example,

```
(defun foo (x)
   (declare (type single-float-pvar x))
   (+!! x (!! 2.0)))
```

may be rewritten as

```
(defun foo (x)
   (declare (type single-float-pvar x))
   (+!! x 2.0))
```

Both the *Lisp interpreter and the *Lisp compiler implement this new behavior. Version F18 of the *Lisp simulator, distributed with Version 6.0 of *Lisp, also includes this feature.

Scalar promotion is enabled by default. It may be disabled in the interpreter, in the compiler, and also in the simulator, by modification of the appropriate global variable.

To disable conversion of scalar arguments:

- in the *Lisp interpreter, set the variable ***lisp–i::*convert–scalar–args–p*** to **nil**
- in the *Lisp compiler, set the variable **slc::*promote–scalars*** to **nil**
- in the *Lisp simulator, set the variable ***lisp–i::*convert–scalar–args–p*** to **nil**

A pair of utility functions is provided that enable/disable the scalar promotion feature:

- to enable scalar promotion, call the function **(*lisp–i::enable–scalar–promotion)**
- to disable scalar promotion, call the function **(*lisp–i::disable–scalar–promotion)**

A small number of *Lisp operators that accept pvars as arguments do *not* automatically promote scalars to pvars. Most of these operators do not accept temporary constant pvars as arguments, and therefore cannot accept scalar constants as arguments. A few operators, in particular ***apply**, and ***funcall**, accept scalar constants as arguments and therefore cannot unambiguously promote scalars to pvars.

The following operators *do not* automatically promote scalars to pvars.

!!	**alias!!**	***apply**
array–to–pvar	**array–to–pvar–grid**	**create–segment–set!!**
***deallocate**	**describe–pvar**	***funcall**
***nreverse**	***processorwise**	**pvar–exponent–length**
pvar–length	**pvar–location**	**pvar–mantissa–length**
pvar–name	**pvar–plist**	**pvar–type**
pvar–vp–set	**pvarp**	***sideways–array**
sideways–array–p	***slicewise**	**typep!!**

User-defined functions may also require the use of **!!** to supply constant pvar arguments, in particular functions that pass their arguments to a *Lisp operator that does not perform scalar promotion.

Also, if an argument to a compiled user-defined function is declared to be a pvar, scalar values cannot be provided for that argument. The **!!** operator must be used in this case.

For example,

```
(defun foo (x y)
   (declare (type (field-pvar 32) x y))
   (+!! x y))

(foo 3 4)
```

will fail because the *Lisp compiler generates Paris code that assumes **x** and **y** are pvars and (at other than zero safety) emits error-checking code to determine whether **x** and **y** are really field pvars.

One other limitation is that 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!!)) ... )
```

4.2 New Options for *Lisp Operators

This section describes the new options for *Lisp operators provided in Version 6.0. These include:

- New keyword arguments for the *Lisp operators **scan!!** and ***pset**
- A new combiner keyword added to ***pset**
- A global variable to allow control of run-time context selection performed by **pref**

4.2.1 New :segment-mode Keyword Argument for scan!!

A new keyword argument, **:segment-mode**, has been added to **scan!!**. Its value can be either **:start**, **:segment**, or **nil**. This argument controls whether the **:segment-pvar** argument is evaluated in all processors or only within the currently selected set.

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.

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** 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 *Paris Reference Manual.*

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-address-length*)
               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).

4.2.2 New :combine–with–dest Keyword Argument for *pset

A new keyword argument, **:combine–with–dest**, has been added to ***pset**. If provided, it specifies that the values already existing in the destination pvar are combined with the values being sent from the source processors. Before Version 6.0, the only behavior possible was to overwrite the values in the destination processors.

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
```

4.2.3 New :queue Combiner Argument for *pset

A new *combine–method* argument, **:queue**, has been added to ***pset**. 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.

4.2.4 Global Variable Control of Run-time Context of pref Function

Prior to Version 5.0 of *Lisp, the **pref** operator evaluated its *pvar–expression* argument in all active processors, both in the interpreter and when compiled. In Version 5.0 of *Lisp, the interpreter was modified so that **pref** would evaluate its *pvar–expression* argument only in the single processor selected by the *send–address* argument. However, the compiler was not modified, so compiled calls to **pref** would still evaluate *pvar–expression* in the active processors.

In Version 6.0, the interpreted version of **pref** has been modified so that it again evaluates its *pvar–expression* argument in all active processors, conforming to the correct semantics of the compiled version. However, for those users whose code depends on the Version 5.0 semantics of **pref**, a global variable, ***lisp–i:\*pref–subselects–processors***, has been added to the interpreter that allows run-time selection of the context in which **pref** evaluates its *pvar–expression* argument.

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.

4.3 New Functions in *Lisp Package

This section describes functions that have been added to the *Lisp package in Version 6.0.

4.3.1 New Vector Pvar Operators

The following vector pvar operations have been added to both the interpreter/compiler and the simulator:

v+scalar!!	**v–scalar!!**	**v*scalar!!**	**v/scalar!!**
vector–normal!!			

These operations are general versions of the following vector pvar functions, which are obsolete as of Version 6.0:

sf–v+–constant!!	**sf–v—constant!!**	**sf–v*–constant!!**
sf–v/–constant!!	**sf–vector–normal!!**	
dsf–v+–constant!!	**dsf–v—constant!!**	**dsf–v*–constant!!**
dsf–v/–constant!!	**dsf–vector–normal!!**	

4.3.2 Internal deallocate–geometry Function Now External

The previously internal function **deallocate–geometry** is now accessible from the *Lisp package.

4.3.3 New *Lisp Dictionary Pages

Dictionary entries for the above functions are included here.

These pages may be added to the **Lisp Dictionary* as follows:

- The **deallocate–geometry** entry should by inserted between pages 336 and 337.
- The **v{+–*/}scalar!!** entry should be inserted between pages 1026 and 1027.
- The **vector–normal!!** entry should be inserted between pages 996 and 997.

deallocate–geometry *[Function]*

Deallocates an existing geometry object.

Arguments

deallocate–geometry *geometry*

geometry	Geometry object. Geometry to be deallocated.

Returned Value

nil	Evaluated for side effect.

Side Effects

The geometry specified by *geometry* is deallocated.

Description

This function is new as of *Lisp Version 6.0.

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.

v+scalar!!, v–scalar!! v*scalar!!, v/scalar!! [*Function*]

Perform an elementwise arithmetic operation on a vector pvar.

Arguments

v+scalar!! *vector–pvar scalar–pvar*

v–scalar!! *vector–pvar scalar–pvar*

v*scalar!! *vector–pvar scalar–pvar*

v/scalar!! *vector–pvar scalar–pvar*

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

These functions are new as of *Lisp Version 6.0.

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

These functions are part of a group of related vector pvar operators, listed below:

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

vector–normal!! [*Function*]

Calculates in parallel the normalized cross-product of the supplied vector pvars.

Arguments

vector–normal!! *vector–pvar–1 vector–pvar–2*

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 is new as of *Lisp Version 6.0.

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

4.4 *Lisp Language Restrictions Update

Most previously reported *Lisp language implementation errors and restrictions have been corrected for the release of *Lisp Version 6.0. The known outstanding bugs and restrictions are reported again in these release notes. All past issues of Programming in *Lisp *In Parallel* and all previous **Lisp Release Notes* may therefore be discarded.

4.4.1 Known Errors Corrected

The following implementation errors reported in *In Parallel* Vol. 3, No. 1, March 1990, are fixed in *Lisp Version 6.0:

allocate–vp–set–processors–bug
def–vp–set–with–eval–when
news–direction–bug
no–heap–or–stack–pvar–symbol–recycling
ppp–address–object–grid–bug
ppp–css–bug
ppp–ordering–bug
sf–v–interpreted–bug
***defvar–error–on–locked–package–symbol**

The following previously unreported implementation errors are fixed in *Lisp Version 6.0:

ID pref–send–address–too–large–bug

Environment

*Lisp, Version 5.2

Description

When **pref!!** is called with a *send–address–pvar* argument that has been declared to be of a size larger than the send-address length of the destination VP set, an error is signalled if any values in the pvar are larger than the send-address length, even if those values are deselected by the currently selected set.

Reproduce By

```
(def-vp-set creatures '(16384)
   :*defvars ((active (evenp!! (send-address!!))
               nil boolean-pvar)))

(*with-vp-set creatures
   (*when active?
      (*let ((address (!! 3)))
         (declare (type (unsigned-byte-pvar 15) address))
         (pref!! (!! 1) address :vp-set creatures))))

>>Error: In interpreted *PSET.
The cube address or source expression in *pset is too big.
There are 16384 selected processors, 16384 processors have
an error. A pvar of type (UNSIGNED-BYTE 15) named ADDRESS
caused the error.
```

Workaround

Declare the *send–address–pvar* argument to be of the same size as the send-address length of the destination VP set, or initialize the address argument so that no value of the pvar (selected or deselected) is illegal for the destination VP set. For example:

```
(*with-vp-set creatures
   (*when active?
      (*let (address)
         (declare (type (unsigned-byte-pvar 15) address))
         (*all (*set address (!! 0)))
         (*set address (!! 3))
         (pref!! (!! 1) address :vp-set creatures))))
```

Status

Patched in 5.2; fixed in 6.0.

ID *defstruct–erroneous–slot–name–warning–bug

Environment

*Lisp, Version 5.2

Description

When ***defstruct** encounters a slot named **p**, it signals a warning that this slot's accessor function's name will conflict with the name of the automatically generated structure predicate, even when the **:conc–name** option has been set to a different symbol than the structure's name.

Reproduce By

```
(*defstruct (foo (:conc-name bar-))
  (p 0 :type fixnum))
;;; Warning: I'll bet you really don't want a slot named
;;; 'P', because this will conflict with the name of the
;;; predicate function, FOO-P.
```

Workaround

Use some other slot name than **p**.

Status

Patched in 5.2; fixed in 6.0.

4.4.2 Known Errors and Restrictions

All known language restrictions for Version 6.0 *Lisp operation are reported here in alphabetical order by bug report ID. If new bugs are discovered, they will be reported during the coming months in the *In Parallel* software bulletin, Vol. 4.

ID star-defstruct-redefinition-bug

Environment

*Lisp, Version 5.2, 6.0; any front end; any CM configuration.

Description

Redefining a parallel structure results in a *Lisp compiler error in one particular case.

Suppose a parallel structure named **plugh** is defined (using ***defstruct**) with two slots, **a** and **b**. Further suppose that **plugh** is then redefined without the **b** slot. Now, if an independent function that happens to be called **plugh-b!!** is also defined, then an attempt to *compile a call to **plugh-b!!** causes the *Lisp compiler to generate internal inconsistency errors.

Reproduce By

```
(*defstruct plugh (a nil :type boolean)
                  (b nil :type boolean))
(*defstruct plugh (a nil :type boolean))
(defun plugh-b!! (x) (1+!! x))
(defun bug (dest source)
   (*set (the boolean-pvar dest)
      (plugh-b!! (the (pvar plugh) source))))
(compile 'bug)
Warning (not associated with any definition):
  Internal inconsistency, assumption failed, while compiling
  (the (pvar plugh) source). Trying to compile a structure
  accessor, but I don't know what type it is [...]
```

Workaround

Set the property list of the function name to **nil**:

```
(setf (symbol-plist 'plugh-b!!) nil)
```

Status

Open.

5 *Lisp Interpreter Version 6.0

The *Lisp interpreter runs on top of either Lucid Common Lisp or Symbolics Common Lisp and executes *Lisp code on the Connection Machine in an interpretive manner.

5.1 Interpreter Restrictions

The following restrictions, which existed in previous versions, still apply in Version 6.0:

- The Common Lisp functions **proclaim** and **setf** are still redefined by *Lisp. This has caused problems in a *Lisp environment on a Symbolics Lisp machine with compilation of Lisp files that are independent of *Lisp and that have been subsequently loaded into an environment without *Lisp. In a future release, **proclaim** and **setf** may no longer be redefined by *Lisp and this problem will no longer exist.

- Several functions that take integer arguments are restricted in that the arguments may not exceed the length of **cm:*maximum–integer–length***. These functions are **isqrt!!**, **float!!**, ***!!**, **floor!!**, **truncate!!**, **ceiling!!**, **round!!**, **mod!!**, and **rem!!**. This problem occurs in both the interpreter and the compiler; it reflects Paris restrictions.

- For segmented scans, as for non-segmented scans, the floating-point numbers scanned are normalized with respect to the maximum value in the entire pvar, across all segments. They are not normalized with respect to the maximum value within a segment only. As a result, the values for scans computed for certain segments—those with values of a much smaller order of magnitude than the maximum—may be lost entirely. Only segments containing values of the same order of magnitude as the maximum value across all segments will have meaningful results.

6 *Lisp Compiler Version 6.0

The *Lisp compiler is compatible with and executes as part of the Common Lisp compiler. Virtually all *Lisp operations can be compiled when properly declared; those that cannot be compiled run interpreted. For *Lisp operations that *are* compiled, the *Lisp compiler generates compiled Lisp/Paris code that runs more efficiently than interpreted *Lisp.

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 *Lisp compiler (*x* is *compiled) is distinguished from compilation by the Common Lisp compiler (*x* is compiled).

This section lists enhancements, corrections, and restrictions to the compiler in Version 6.0.

6.1 *Lisp Compiler Enhancements

The following enhancements distinguish Version 6.0 of the compiler from previous versions:

- **More Complete Compilation of *Lisp Code**. The *Lisp compiler now compiles virtually all the functions and macros of the *Lisp language when proper pvar type declarations are provided.

- **Faster Execution of *Lisp Code**. *Lisp code should execute faster, because of improved code generation and many improvements in performance at the Paris level. For details of these improvements, see the chapter on Paris in the *CMSS Summary* for Version 6.0.

6.2 *Lisp Compiler Limitations

This section describes the current limitations on the use of the *Lisp compiler to *compile *Lisp code.

6.2.1 *Lisp Operations That Don't *Compile

The following *Lisp operations do not *compile as of Version 6.0. (This list supersedes all such lists included in the **Lisp Release Notes* for previous versions.)

The following *Lisp operations do not compile as yet:

address–nth!!	**address–plus–nth!!**	**address–rank!!**

Parallel sequence operations do not *compile. These operations are listed below. The one exception is the **reduce!!** operation, which does *compile in limited cases (see Section 6.2.3).

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!!	**some!!**	**subseq!!**
substitute!!	**substitute–if!!**	**substitute–if–not!!**

The following segment set scanning operations do not *compile:

create–segment–set!!	**segment–set–scan!!**

6.2.2 Obsolete *Lisp Operations

The following functions were introduced to provide optimized interpreter performance for floating-point vector operations. The *Lisp compiler now *compiles the corresponding general vector operators. As of Version 6.0, the following operators are therefore obsolete:

dsf–cross–product!!	**dsf–vector–normal!!**
dsf–vscale–to–unit–vector!!	**sf–cross–product!!**
sf–v+–constant!!	**sf–v—constant!!**
sf–v*–constant!!	**sf–v/–constant!!**
sf–vabs!!	**sf–vabs–squared!!**
sf–vector–normal!!	**sf–vscale–to–unit–vector!!**

6.2.3 *Lisp Compiler Restrictions

The current version of the *Lisp compiler will *compile virtually all *Lisp operations, when proper type declarations are provided. Anything that is not *compiled is handled by the interpreter. If the **Warning Level** compiler option is set to **High**, the *Lisp compiler prints a warning whenever an operation is not *compiled.

The following *Lisp operations *compile with specific restrictions.

ash!! [*Function*]

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 is 2^32 this operation would produce 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**).

code–char!! **int–char!!** **make–char!!** [*Function*]

These operations will only *compile when used as an argument to a *Lisp operation that expects a character pvar, such as ***set** or **character!!**.

digit–char–p!! [*Function*]

This operation only *compiles when used as an argument to a *Lisp operation that expects an integer pvar as its argument, such as ***set**.

grid!! **grid–relative!!**

These operations will *compile only in restricted circumstances, specifically when they are used as arguments to the ***pset** or **pref!!** operators.

pref!! [*Macro*]

If the *pvar–expression* argument is not a simple expression, such as a variable, this operation will only *compile when the **:vp–set** argument is either unspecified or ***current–vp–set***.

***pset** *[Macro]*

The ***pset** operation will not *compile with the **:default** *combine–method* argument; use the **:no–collisions** option instead. Also, ***pset** does not yet *compile with the **:queue** *combine–method* argument.

reduce!! *[Function]*

The **reduce!!** operation will not *compile if given a user-defined function as its *function* argument, and also will not *compile if any of its keyword arguments are specified.

scan!! *[Function]*

This operation will *compile only if the *function* argument is one of the specialized scanning operators such as **+!!**, **max!!**, **min!!**, etc. If the *function* argument is ***!!**, then **scan!!** will *compile, but only if *pvar* is a floating-point pvar.

6.2.4 Special Forms That *Compile

The following special forms are recognized and handled by the *Lisp compiler:

compiler–let **let** **let*** **progn**

6.3 Type Declarations and the *Lisp Code Walker

The *Lisp compiler is enabled by default. The *Lisp compiler can translate virtually all *Lisp statements into compiled Lisp/Paris. Any *Lisp statement that cannot be translated is interpreted by the *Lisp interpreter.

The *Lisp compiler includes a code walker that allows more complete compilation of *Lisp code. The code walker is enabled by default, and is described in more detail below.

The key to effective use of the *Lisp compiler is complete and correct declaration of *Lisp code. Users of the *Lisp compiler will want to consult Chapter 4, "*Lisp Type Declaration," in the **Lisp Dictionary*, for guidelines and examples of proper declaration of *Lisp code.

An additional type declaration issue, omitted from the **Lisp Dictionary*, is described in Section 6.3.2.

6.3.1 The Code Walker

The *Lisp compiler includes a code walker, which allows the compiler to compile *Lisp code more thoroughly. The code walker is an extension of the *Lisp compiler that "walks" through all the individual forms of a piece of *Lisp code. It records all declarations it encounters and compiles each *Lisp form it finds. The code walker can be enabled and disabled by the user. It is enabled by default.

The code walker allows the *Lisp compiler to:

- Find declarations it would otherwise ignore.
- Generate *compiled code for *Lisp expressions that would not otherwise be *compiled.

The *Lisp code walker is an extension of the CommonLoops code walker developed at Xerox Palo Alto Research Center. CommonLoops, including its code walker, is generously made available by Xerox Corporation to the Common Lisp community for the preparation of derivative works.

6.3.1.1 Effect of Code Walker on Code Compilation.

The *Lisp compiler has these capabilities when the code walker is enabled:

- *Lisp declarations are recognized in all locations where Common Lisp allows declaration forms. In particular, the *Lisp compiler can now recognize declarations within **defun**, **let**, and **let*** forms without the need to use the ***locally** construct.
- All properly declared *Lisp forms are *compiled, not only those within the scope of a *Lisp macro operator such as ***set**.

6.3.1.2 Enabling the Code Walker

The code walker can be enabled and disabled by the user. It is enabled by default. To enable the code walker, do either of the following:

1. Type `(compiler-options)`

   This will display a menu of compiler options. At the bottom of the menu is an item that enables/disables the code walker.

2. Set the variable ***lisp:*use–code–walker*** to **t**. For example,

   ```
   (setq *lisp:*use-code-walker* t)
   ```

   enables the code walker for all *Lisp code that is *compiled. Setting ***lisp:*use–code–walker*** to **nil** disables the code walker.

6.3.1.3 Using the Code Walker: An Example

With the code walker disabled, if one wanted to write a function that *compiled, one would need to write it like this:

```
(defun sum-of-squares!! (x y)
  (*locally          ;;; *locally to declare arguments x and y
     (declare (type single-float-pvar x y))

     (*let (result)  ;;; declaration of result within *let form
       (declare (type single-float-pvar result))

       (*set result (+!! (*!! x x) (*!! y y)))

       result)))
```

With the code walker enabled, the *Lisp compiler recognizes declarations in all the places Common Lisp permits declarations, without the need for ***locally**. In particular, the *Lisp compiler recognizes declarations within **defun**, **let**, and **let*** forms. A list of all special forms within which Common Lisp permits declarations may be found in *Common Lisp: The Language*, Second Edition, pp. 215–16.

Thus, with the code walker enabled one can write the **sum–of–squares** definition as

```
(defun sum-of-squares!! (x y)
  (declare (type single-float-pvar x y))

  (*let (result)
    (declare (type single-float-pvar result))

    (*set result (+!! (*!! x x) (*!! y y)))

    result))
```

In addition, the code walker enables the *Lisp compiler to *compile all properly declared *Lisp forms, not just those within the scope of a *Lisp macro operator such as ***set**. Because of this change, the **sum-of-squares** definition may be condensed even further, producing

```
(defun sum-of-squares!! (x y)
  (declare (type single-float-pvar x y))
  (+!! (*!! x x) (*!! y y)))  ;;; *let and local variable result
                              ;;;  are no longer needed
```

which will also *compile into Paris code.

6.3.1.4 Automatic Declaration of Loop Indices

A minor feature of the code walker is that it automatically declares iteration variables as integers, eliminating the need for separate declaration of these variables. For example, with the code walker disabled,

```
(dotimes (j 100)
  (*set x (*!! x (!! j))))
```

will not *compile unless **(!! (the fixnum j))** is used instead of **(!! j)**. With the code walker enabled, special declarations are unnecessary and this code will *compile as is.

6.3.2 Proper Declaration of *defun Forms

When a function defined by the *Lisp operator ***defun** is referenced prior to its definition in a file of code that is intended to be compiled, a special declaration form is required for the compiler to handle the reference correctly.

The ***defun** macro actually defines two operators: a function that performs the operations specified by the body of the ***defun**, and a macro that calls this function as well as performing other tasks related to reclamation of space on the CM stack. The macro is given the name specified by the ***defun** form, and the function is given a name derived from the name of the macro. For example, if ***defun** is used to define an operator called **marbles**, the macro will be named **marbles**, but the function will be named ***defun-marbles.**

If a ***defun** is referenced prior to its definition in a file, then the Lisp compiler will not treat the reference as a macro call (as the user might intend), but will instead compile it as a call to an ordinary function. The "external" operator defined by ***defun** is a macro rather than a function, and thus function calls compiled prior to the ***defun** form will signal an error when executed.

To avoid this problem, place a ***proclaim** form in the file prior to all references to the operator defined by ***defun**. The ***proclaim** macro recognizes a special keyword, ***defun**, that can be used to "forward reference" an operator that will eventually be defined by ***defun**.

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))
```

Note the use of ***proclaim** to declare the data type of the pvar returned by **xyzzy-foo**. The ***proclaim** macro can be used to declare the data type of a ***defun** operator, but only if the operator has previously been declared as a ***defun** with ***proclaim**, as shown in this example.

6.4 Viewing *Compiled Code

It is possible to have the *Lisp compiler display *compiled code produced during compilation of a *Lisp expression. The value of the compiler global variable **slc::*show-expanded-code*** determines whether this feature is active.

When the value of **slc::*show-expanded-code*** is **t**, the *Lisp compiler will print out any *compiled code produced whenever a *Lisp form is compiled.

When the value of **slc::*show-expanded-code*** is **nil**, the default, this feature is disabled.

For example, given the function definition

```
(defun hyp!! (x y)
   (declare (type single-float-pvar x y))
   (sqrt!! (+!! (*!! x x) (*!! y y))))
```

when **hyp!!** is compiled, as in

```
(setq slc::*show-expanded-code* t
      *safety* 0
      *use-code-walker* t)

(compile 'hyp!!)
```

the following output is displayed:

```
expression:

(SQRT!! (+!! (*!! X X) (*!! Y Y)))

expanded to:

(LET*  ( (SLC::STACK-FIELD (CM:ALLOCATE-STACK-FIELD 32))
         (#:SQRT!!-INDEX-2 (+ SLC::STACK-FIELD 32)))
   (DECLARE (TYPE SLC::CM-ADDRESS SLC::STACK-FIELD
                                  #:SQRT!!-INDEX-2))
   (DECLARE (IGNORE #:SQRT!!-INDEX-2))
   (CM:F-MULTIPLY-3-1L
    SLC::STACK-FIELD (PVAR-LOCATION Y) (PVAR-LOCATION Y) 23 8)
   (CM:F-MULT-ADD-1L
    SLC::STACK-FIELD (PVAR-LOCATION X) (PVAR-LOCATION X)
    SLC::STACK-FIELD 23 8)

   ;;  - sqrt!!.
   (CM:F-SQRT-2-1L SLC::STACK-FIELD SLC::STACK-FIELD 23 8)
   (SLC::ALLOCATE-TEMP-PVAR
      :TYPE :FLOAT :LENGTH 32 :BASE SLC::STACK-FIELD
      :MANTISSA-LENGTH 23 :EXPONENT-LENGTH 8))
```

6.5 Compiler Options Notes

6.5.1 Compiler Options Function

The **slc::report–options** function may be used to view the current settings of *Lisp compiler options.

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

On Lucid Common Lisp or Sun Common Lisp this function takes an optional argument, which if non-**nil** reports the Lucid (Sun) compiler options as well.

```
(slc::report-options t) ; To see Lucid (Sun) compiler options
```

6.5.2 Additional Compiler Options

Compiler options not described in the **Lisp Compiler Guide* are described here. Note that these options are only visible on the menu displayed by **(compiler–options :class :all)**.

Note also that the following three options apply to Symbolics front end users only:
Macroexpand Repeat **Macroexpand Inline Forms** **Macroexpand Print Case**

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 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.

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))
```

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 **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.

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.

6.5.3 Restrictions on Compiler Options

Some *Lisp compiler options have limited functionality. These options are described below.

Pull Out Common Address Expressions

This option is not fully implemented and therefore should not be used in most cases.

Compilation Speed

Except as a constraint on peephole and binding optimization, this option is currently not considered by the compiler.

Space
Speed

These options are currently not used by the compiler.

6.6 *Lisp Compiler Implementation Errors Update

Most previously reported *Lisp compiler implementation errors have been corrected for *Lisp Version 6.0. The known outstanding bugs and restrictions are reported again in these release notes. All past issues of Programming in *Lisp *In Parallel* and all previous **Lisp Release Notes* may therefore be discarded.

6.6.1 Known Errors Corrected

The following compiler implementation errors reported in *In Parallel* Vol. 3, No. 1, March 1990, are fixed in *Lisp Version 6.0:

defconstant–dim–scan–compiler–bug
float–pvar–initial–value–bug
make–array–not–compiling–with–zero–element
pref–expr–interpreted–semantics–change

The following previously unreported compiler errors are fixed in *Lisp Version 6.0:

ID star–case–compiler–warning

Environment

*Lisp compiler, Version 5.2

Description

The ***case** operator signals a warning about an internally generated symbol being undeclared.

Reproduce By

Compile the following:

```
(defun test ()
  (*case (random!! (!! 100))
    (0 (print (*sum (!! 1))))
    (1 (print (*sum (!! 2))))))
Warning (not associated with any definition):
While compiling #:ONCE-ONLY-VAR3 in function TEST:
The expression (*SET #:RET-4 (NOT!! (NULL!! #)))
is not compiled because the *Lisp compiler cannot find a
type declaration for the symbol ONCE-ONLY-VAR3.
```

Status

Fixed in Version 6.0; patch is available for previous versions.

ID pref–inactive–processors–bug

Environment

*Lisp, Version 5.2

Description

When **pref!!** is *compiled with the **:many–collisions** option, it generates a call to **slc::get–many–collisions**. This operation had a bug that caused it to operate incorrectly when all processors are selected.

Reproduce By

Compile a **pref!!** expression with the **:many–collisions** option.

Workaround

Turn off *Lisp compiler around form or remove **:many–collisions** option.

Status

Patched in 5.2; fixed in 6.0.

6.6.2 Known Errors Still Open

All previously reported unintentional compiler restrictions have been fixed in Version 6.0. If new bugs are discovered, they will be reported during the coming months in the *In Parallel* software bulletin, Vol. 4.

6.7 Miscellaneous Compilation Notes

Additional notes and hints on the use of the *Lisp compiler are included here. These are reprinted from previous release notes.

6.7.1 Non-simple Pvar Expression Compilation in pref!!

The *Lisp compiler can *compile a **pref!!** form that specifies a non-simple *pvar–expression* source argument, if the **:vp–set** keyword argument is either unspecified or given as ***current–vp–set***.

6.7.2 Warnings on Non-compiled Code

Unless the *Lisp compiler **Warning Level** is explicitly set to **High**, the *Lisp compiler will not emit warning messages to the effect that it could not translate certain *Lisp statements into Lisp/Paris. Thus, using the default **Warning Level**, it is not possible to know which portions of code have been translated into Lisp/Paris and which have not.

6.7.3 *Lisp Compiler Warning Level and Safety Level Options

Do not confuse the compiler **Warning Level** with the **Safety Level**. The **Warning Level** determines how completely the compiler reports compile-time problems. The **Safety Level** determines the degree to which compiled code reports run-time errors.

IMPORTANT

The efficient execution of compiled code depends highly on the compiler **Safety Level**. The user is strongly advised to become familiar with the proper setting of different safety levels. Refer to the **Lisp Compiler Guide* for descriptions of these options.

7 *Lisp Simulator Version 6.0

The *Lisp simulator runs on top of Common Lisp and allows users to execute *Lisp code without using a Connection Machine system. The *Lisp simulator is known to run on the following implementations of Common Lisp:

- Symbolics Lisp on a Symbolics Lisp machine
- Sun Common Lisp on a Sun-4 Workstation
- Lucid Common Lisp on a VAX running ULTRIX

The *Lisp simulator has also been known to run on the following systems:

- Sun Common Lisp on a Sun-3 Workstation
- Lucid Common Lisp on a VAX running VMS
- Lucid Common Lisp on an Apollo workstation
- Coral Common Lisp on a Macintosh
- Kyoto Common Lisp on various machines

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

7.1 New *Lisp Simulator Version

Version F18 of the *Lisp simulator, released along with Version 6.0 of the *Lisp software, is fully compatible with *Lisp Version 6.0.

7.2 *Lisp Simulator Now Freely Available

Thinking Machines Corporation is pleased to announce that the *Lisp simulator is now freely available for use, copying, and modification. You are free to distribute and modify the *Lisp simulator without restriction.

The *Lisp simulator is available via anonymous FTP in the **/public** directory of **think.com**. The file containing the simulator is a UNIX "shar" file called **starsim–f18–sharfile** (where **f18** may be replaced in the future by a higher release number).

This sharfile provides the necessary sources and systems for the *Lisp simulator to run under Symbolics, Lucid, Allegro, and Franz Common Lisp. Porting the *Lisp simulator to other Common Lisp implementation is generally a simple matter.

NOTE: If you do port the *Lisp simulator to a version of Common Lisp other than those listed above, we ask that you send a description of any required modifications to Thinking Machines Corporation Customer Support, so that these changes can be incorporated into future versions of the simulator.

People wishing to distribute the *Lisp simulator should distribute it from the sharfile described above, and not from the sources provided on-site at Connection Machine customer installations. The sharfile includes documentation, instructions, and auxiliary files that are useful in installing the *Lisp simulator at non–Connection Machine sites.

Thinking Machines will continue to provide support for the *Lisp simulator for Connection Machine system customers. Thinking Machines is under no obligation to provide support to other users of the *Lisp simulator, either in porting or use of the simulator.

7.3 *Lisp Simulator Restrictions Update

Most previously reported *Lisp simulator implementation errors and restrictions have been corrected for the release of *Lisp Version 6.0. The known outstanding bugs and restrictions are reported again in these release notes. All past issues of Programming in *Lisp *In Parallel* and all previous **Lisp Release Notes* may therefore be discarded.

7.3.1 Restriction on Pvar Types

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

7.3.2 Known Errors Corrected

The following simulator implementation errors reported in *In Parallel* Vol. 3, No. 1, March 1990, are fixed in Version F18 of the *Lisp simulator:

create–vp–set–not–for–uninstantiated
parallel–vector–and–array–sim–omission

ppp–ordering–sim–bug
rank–sort–neg–fg–data–sim–bug
star–setf–of–row–major–sideways–aref–sim–bug

7.3.3 Known Errors and Restrictions

All known unintentional simulator restrictions for Version F18 of the simulator are reported here, in alphabetical order by bug report ID. If new bugs are discovered, they will be reported during the coming months in the *In Parallel* software bulletin, Vol. IV.

ID lucid–exit–from–sim–bug

Environment

*Lisp simulator Version F16, F17, F18; Lucid Common Lisp

Description

When running the *Lisp simulator under either Lucid Lisp or Sun Lisp, attempting to access a deallocated pvar causes the lisp process to quit.

Reproduce By

```
(*cold-boot)
(setq a (allocate!! (!! 0)))
#<Structure PVAR C0B1CE>

(*cold-boot)
(ppp a)

End of File read by debugger -- quitting Lisp
```

Workaround

Attempting to access a deallocated pvar is an error.

Status

Open.

7.4 Notes on Simulator Use

7.4.1 Porting Code

*Lisp code can be ported from the simulator to the interpreter/compiler (and vice versa) with few modifications. However, *all* code *must* be recompiled when porting in either direction.

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

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

7.4.3 Conditional Simulator Compilation and Execution

It is sometimes desirable to write *Lisp code in one fashion to execute on a Connection Machine system and in another fashion to execute in the *Lisp simulator. This is especially helpful where code intended to execute on the Connection Machine hardware uses different constructs and definitions than code intended for the simulator.

To signal the Lisp reader to conditionally read a form depending on whether or not the *Lisp simulator is loaded, use the Common Lisp **#+** reader macro with the feature symbols ***LISP–SIMULATOR** and ***LISP–HARDWARE.**

Thus,

#+*LISP–SIMULATOR *form*

reads *form* only if the *Lisp simulator is loaded.

#+*LISP–HARDWARE *form*

reads *form* only if *Lisp is loaded and a Connection Machine system is attached to the executing front-end computer. For example, the expression

```
(progn
   #+*LISP-HARDWARE
      (*cold-boot :initial-dimensions  '(256 256 4))
   #+*LISP-SIMULATOR
      (*cold-boot :initial-dimensions  '(8 8 2)))
```

will execute properly both on the Connection Machine hardware and in the *Lisp simulator. The ***LISP–HARDWARE** symbol is used to select a large VP set when the Connection Machine hardware is available. The ***LISP–SIMULATOR** symbol is used to select a smaller VP set when the *Lisp simulator is in use.

Note that it is possible to conditionalize individual components of a function call using these feature symbols. This is useful in those cases where the expression to be conditionalized is very long or complex, and it is therefore desirable for purposes of code support not to have two separate, independently conditionalized copies of the expression.

For example, the ***cold–boot** example given above can be rewritten in the following form:

```
(*cold-boot :initial-dimensions  #+*LISP-HARDWARE  '(256 256 4)
                                 #+*LISP-SIMULATOR '(8 8 2))
```

There is also a version symbol available, ***LISP–SIMULATOR–F18**, that may be used to conditionalize code that should be executed only in the F18 version of the *Lisp simulator. This symbol may be used in a similar manner as those shown above.

For example, the scalar promotion feature is available in Version F18 of the simulator, but not in Version F17. The following code will execute properly in either version of the simulator:

```
(*set dest  (+!!  #+*LISP-SIMULATOR-F18 3
                  #-*LISP-SIMULATOR-F18 (!! 3) x))
```

8 *Lisp Library Version 6.0

The *Lisp Library is a set of *Lisp functions and macros made available in the form of an on-line software library. Please note that all code included in the library is experimental. Users are welcome to make use of the library code at their own risk, with the understanding that some or all of these functions and macros may not be supported in future releases.

8.1 Changes for Version 6.0

As of Version 6.0, the following module has been added to the *Lisp library:

• PVAR–IO	Read and write pvars to front-end disks.

This module contains a set of functions that can be used to perform fast data transfer between a front-end disk and the CM.

8.2 Accessing the *Lisp Library

The *Lisp library code is available in the directory

```
/cm/starlisp/library/f6000/*
```

On-line documentation for the library functions and macros is available in the file

```
/cm/starlisp/library/f6000/documentation.text
```

Ask your systems administrator or applications engineer to help you locate these files at your site.

All functions in the library are defined to autoload on demand. When any one function in a given interface file is autoloaded, the rest of the functions in that interface file are also autoloaded.

8.3 *Lisp Library Contents

The following interface files are included in the *Lisp library in Version 6.0:

• AREF32–SHARED	Lookup table interface.
• FAST–RNG	Fast random number generator.
• ROW–AND–COLUMN–MAJOR	Row/column major address interface.
• LET–ALIAS	Temporary storage reduction tool.
• COLLECTED–MACROS	Useful macros.
• PVAR–IO	Read and write pvars to front-end disks.
• FFT	CMSSL Fast Fourier Transform interface.
• MATRIX–MULTIPLY	CMSSL matrix multiplication interface.

8.4 *Lisp Library Restrictions Update

All previously reported *Lisp Library implementation errors and restrictions have been corrected in the release of *Lisp Version 6.0.

8.4.1 Known Errors Corrected

The following simulator implementation errors reported in *In Parallel* Vol. 3, No. 1, March 1990, are fixed in Version 6.0 of the *Lisp Library:

create–lookup–table–sll–bug

9 *Graphics Version 6.0

*Graphics is a *Lisp interface to the CM graphics programming environment. Documentation for *Graphics can be found in the **Graphics Reference Manual*, which is distributed as part of the volume entitled *Programming in *Lisp* in the Connection Machine documentation set. Information about modifications and corrections made to *Graphics may be found in the Version 6.0 release notes for this document.

Note: The *Graphics documentation has moved. In previous versions this document was distributed as part of the volume entitled *Connection Machine Graphics Programming*. As of Version 6.0 it is included with the *Lisp reference documentation. Please make a note of this change.

10 Fast Graph

Fast Graph is a software package designed to allow optimized router communications for fixed router patterns, and is accessible from *Lisp.

For a communications pattern that is fixed with respect to machine size, VP ratio, and data paths, the Fast Graph package offers the possibility of significantly faster execution than can be obtained through the *Lisp operators ***pset** and **pref!!**.

The Fast Graph package is available from TMC Customer Support or from your applications engineer as unsupported software.

Sample *Lisp code that creates data patterns and executes compiled data patterns is provided with the Fast Graph package.