The
Connection Machine
System

*Lisp Release Notes

Version 6.1
October 1991

These release notes replace all previous release notes

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, October 1991

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

Contents

About These Release Notes

Objectives of This Manual

The **Lisp Release Notes* Version 6.1 describe new and changed features of *Lisp introduced with Connection Machine System Software Version 6.1. The **Lisp Release Notes* also provide useful programming information not contained in other *Lisp documentation.

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 (CM) system.

Revision Information

These release notes are new with *Lisp Version 6.1, and *replace all previous release notes.*

Organization of These Release Notes

1. **About Version 6.1**
   Describes the *Lisp system, the current *Lisp documentation set, and summarizes the changes and enhancements made to *Lisp in Version 6.1.

2. **Porting to Version 6.1**
   Explains how to port *Lisp code developed in versions prior to Version 6.1, and provides a list of obsolete *Lisp language features.

3. ***Lisp Documentation**
   Describes new, current, and obsolete *Lisp documentation as of Version 6.1.

4. ***Lisp Language Version 6.1**
   Describes language features that are new and enhanced in Version 6.1, and lists known *Lisp language implementation errors.

5. ***Lisp Interpreter Version 6.1**
   Lists known *Lisp interpreter restrictions.

6. ***Lisp Compiler Version 6.1**
   Describes compiler features that are new and enhanced in Version 6.1, and lists known *Lisp compiler implementation errors and restrictions.

7. ***Lisp Simulator Version 6.1**
   Describes simulator features that are new and enhanced in Version 6.1, and lists known *Lisp simulator implementation errors and restrictions.

8. **Sun and Lucid Common Lisp**
   Describes the Lucid Common Lisp environments needed to run *Lisp on Sun-4 and VAX front ends, and lists known Lucid-related implementation errors.

9. ***Lisp Library Version 6.1**
   Describes updates to library of *Lisp source code in Version 6.1.

10. ***Graphics Version 6.1**
    Describes *Lisp interface to the CM graphic programming environment.

11. **Fast Graph**
    Describes Fast Graph grid communication optimization package.

Related Manuals

- *Getting Started in *Lisp*

  This manual provides a tutorial introduction to the *Lisp language, and includes the information you need to get started in writing, compiling, and debugging *Lisp programs. Appendixes provide basic introductions to the CM and to important Paris functions.

- **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, a guide to using type declarations in *Lisp, and an extensive list of *Lisp compiler options. Except as noted in these release notes, the *Dictionary* is the most accurate and current description of the *Lisp language.

- *CM User's Guide*

  This document provides helpful information for users of the Connection Machine system, and includes a chapter on the use of *Lisp on the CM, including information about how to call Paris functions from *Lisp programs.

- *Connection Machine Parallel Instruction Set (Paris)*

  The Paris manual describes the low-level parallel instruction set of the Connection Machine system. *Lisp programmers who want to make use of Paris calls in their programs should also refer to this manual.

- *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 this manual are described below.

Convention	**Meaning**
boldface	*Lisp language elements, such as **:keywords**, functions, macros, pathnames, etc., where they appear embedded in text.
italics	Argument names and placeholders in descriptions of function syntax.
`typewriter`	Code examples and code fragments not embedded in text.
> **`(user-input)`** `lisp-output`	In interactive examples, what you type is in **`bold typewriter`**, and what *Lisp displays in return 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.

If your site has an Applications Engineer or a local site coordinator, please contact that person directly for support. Otherwise, please contact Thinking Machines' home office customer support staff:

U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142–1264
Internet Electronic Mail:	customer–support@think.com
uucp Electronic Mail:	ames!think!customer–support
Telephone:	(617) 234–4000 (617) 876–1111

For Symbolics Users Only

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

`To:` customer–support@think.com

Please supplement the automatic report with any further pertinent information.

1 About *Lisp Version 6.1

The *Lisp language is a parallel extension of Common Lisp for the Connection Machine® massively parallel computing system. *Lisp programs are written in Common Lisp, and include calls to *Lisp operations that control the CM.

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 Sun-4 Workstations running the UNIX operating system, Digital Equipment Corporation VAX machines running the ULTRIX operating system, and Symbolics 3600-series Lisp machines.

The *Lisp simulator can be run on any machine with a Common Lisp language environment. CM hardware is not required to run the simulator.

*Lisp Version 6.1 requires Connection Machine System Software Version 6.1, and runs on the following front-end Lisp software:

- Sun Common Lisp Version 4.0 on Sun-4 front ends
- Lucid Common Lisp Version 2.5 on VAX front ends
- Genera 8.1 on Symbolics 3600-series front ends.

See Section 8 for more information about Sun Common Lisp 4.0 and Lucid Lisp 2.5.

2 Porting to Version 6.1

Version 6.1 is a minor *Lisp release. It includes a number of small language enhancements, and introduces one important new operation, **trace–stack**.

*Lisp source code written in previous versions of *Lisp (5.*n*, 6.0) will run unchanged under Version 6.1. However, *Lisp programs compiled under previous versions must be recompiled to run under Version 6.1.

The following changes distinguish Version 6.1 from previous versions:

New Stack Memory Tracing Function. The utility function **trace–stack** has been added to *Lisp. It can be used to trace CM stack memory usage in a *Lisp program.

New Scalar Function. The function **v/**, a scalar equivalent to **v/!!**. has been added.

New Arguments to Array Copying Functions. The *Lisp functions **array–to–pvar** and **pvar–to–array** now include **:start** and **:end** keywords that are identical to (and replace) the existing **:cube–address–start** and **:cube–address–end** keywords. (These are now obsolete and should not be used.)

New Type Alias. The type alias **fixnum–pvar** can now be used in place of **(pvar fixnum)**.

***Lisp Compiler Code-viewing Macro.** The macro **ppme**, previously an internal function of the compiler, has been made external in the *Lisp package. This macro can be used to view compiled code produced by the *Lisp compiler.

Version 6.1 includes the following changes in software versions and availability:

New Versions of Sun Common Lisp and Genera Software. *Lisp has been updated to use Version 4.0 of Sun Common Lisp and Release 8.1 of Genera.

***Lisp Timesharing Bands Now Available.** *Lisp runs on timeshared CMs, and *Lisp timesharing software bands are available.

New Version of *Lisp Simulator. Version F19 of the simulator is now available.

Version 6.1 *Lisp also includes the following incidental improvements:

Faster Execution of Communication Functions. The *Lisp operations **pref!!** and ***pset** now execute faster, due to improvements at the Paris level.

Faster Image Transfer. *Graphics users will notice faster transfer of images to displays because of improvements of the low-level graphics code.

2.1 Obsolete Language Features

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

As of Version 6.1, the following language features are obsolete:

The archaically named **:cube–address–start** and **:cube–address–end** keyword arguments to **array–to–pvar** and **pvar–to–array** are now obsolete. They have been replaced by the functionally identical keywords **:start** and **:end**.

All obsolete *Lisp operations reported prior to Version 6.1 are listed below, along with the currently existing 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*!!**
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–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

3 *Lisp Documentation

3.1 New Documentation

As of Version 6.1, the following new *Lisp document is available:

- *Getting Started in *Lisp*, Version 6.1, June 1991

*Getting Started in *Lisp* provides a tutorial introduction to the *Lisp language, as well as an overview of the programming, compiling, and debugging tools of the language. The appendixes of this document provide helpful information for new users of the CM.

3.2 Current Documentation

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

- *Getting Started in *Lisp*, Version 6.1, June 1991

  This document is new as of Version 6.1, and is described above.

- The **Lisp Dictionary*, Version 6.1, October 1991

  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, a chapter on *Lisp pvar types and type declaration, and a chapter listing the numerous options of the *Lisp compiler.

- The *CM User's Guide*, Version 6.1, October 1991

  The *Connection Machine System User's Guide* 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, including information on running *Lisp in batch and timesharing modes.

- The **Lisp Release Notes,* Version 6.1, October 1991

  Finally, the **Lisp Release Notes* document all changes to *Lisp as of Version 6.1.

3.3 Obsolete Documentation

The following documents are also available, but have been largely replaced by those listed above.

- 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 Reference Manual* and *Supplement* provide some useful conceptual information on the *Lisp language. However, all reference material in these documents has been superseded by the information contained in the **Lisp Dictionary* and *Getting Started in *Lisp*. Other than as noted in these release notes, the material in the **Lisp Dictionary* is the most current and correct. The **Lisp Compiler Guide* likewise provides useful information about *Lisp compiler, but users of the *Lisp compiler will also want to consult the chapters on type declaration and compiler options in the **Lisp Dictionary*, the chapter describing the compiler in *Getting Started in *Lisp*, and the update information about the compiler included in these *Release Notes*.

3.4 *Lisp On-Line Code Examples

Examples of *Lisp code are available on-line in the following directories. Ask your system administrator or applications engineer to help you locate these files at your site.

```
/cm/starlisp/interpreter/f6100/*example*.lisp
/cm/starlisp/graphics/f6100/examples.lisp
```

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

4 *Lisp Language Version 6.1

4.1 New Features of *Lisp

This section describes changes and additions made to the *Lisp language in Version 6.1.

4.1.1 New Stack Memory Tracing Utility

*Lisp now includes a function, **trace–stack**, that you can use to trace the stack memory usage of your program, and thereby determine what parts of your code are using the most stack memory (that is, creating the most local or temporary pvars). Version 6.1 of the **Lisp Dictionary* includes an entry for **trace–stack** that provides a complete description of this function, along with numerous examples.

4.1.2 New Function and New Function Arguments

The scalar function **v/** has been added, by analogy with **v+** and **v***, to serve as the scalar equivalent of **v/!!**.

The *Lisp functions **array–to–pvar** and **pvar–to–array** now include **:start** and **:end** keywords. These are identical to (and replace) the existing **:cube–address–start** and **:cube–address–end** keywords. The **:cube–address** arguments are obsolete as of Version 6.1, and should not be used in new code.

4.1.3 Faster Execution of Communication Functions

Thanks to improvements in the underlying Paris code, the *Lisp communication operators **pref!!** and ***pset** now execute up to 50% faster, depending on the communication pattern and the current VP ratio. Code run at high VP ratios will most likely obtain the most significant speed improvement.

4.2 Automatic Promotion of Scalar Arguments

This feature of *Lisp was added in Version 6.0. 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))
```

Currently, all 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))
```

This feature is implemented in both the *Lisp interpreter and the *Lisp compiler, as well as in the *Lisp simulator.

4.2.1 Enabling and Disabling Scalar Promotion

Scalar promotion is enabled by default. It may be disabled independently in the interpreter, the compiler, and the simulator by modification of one of three global variables. 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)**

4.2.2 Operations That Do Not Promote Scalar Arguments

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!!**

4.2.3 Other Cases Where Scalar Promotion Does Not Apply

User-defined functions may 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. In these cases the **!!** operator must be used.

For example, if the following function definition is compiled:

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

(compile 'foo)
```

Then a function call such as

```
(foo 3 4)
```

will fail because the *Lisp compiler generates Paris code that assumes **x** and **y** are pvars. In particular, if the *Lisp compiler safety level (***safety***) is not zero, the *Lisp compiler will add error-checking code to determine whether **x** and **y** are really field pvars.

One other case 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!!)) ... )
```

4.3 *Lisp Language Restrictions Update

Most previously reported *Lisp language implementation errors and restrictions have been corrected for the release of *Lisp Version 6.1.

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.

4.3.1 Known Restrictions Still Open

The following restriction is still open:

ID star–defstruct–redefinition–bug

Environment

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

Synopsis

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

Description

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 consistency 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 6.1

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

5.1 Interpreter Restrictions

The following restrictions, which existed in previous versions, still apply in Version 6.1:

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 6.1

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. If the compiler warning level (***warning–level***) is set to **:high**, the *Lisp compiler will signal an error whenever it encounters *Lisp code that cannot be fully compiled.

6.1 *Lisp Compiler Enhancements

This section lists enhancements, corrections, and restrictions to the compiler in Version 6.1.

6.1.1 New Type Alias

The following type declaration has been added:

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

This declaration can be used with all *Lisp declaration operators, including **declare** and ***proclaim**.

6.1.2 Compiled Code Printing Macro Now External

The macro **ppme**, previously an internal function of the compiler, has been made external in the *Lisp package. This macro can be used to view compiled code produced by the *Lisp compiler. The **Lisp Dictionary* entry for **ppme** provides a description and examples of this operator.

6.2 *Lisp Compiler Limitations

This section describes the current limitations of the *Lisp compiler.

6.2.1 *Lisp Operations That Don't Compile

The following *Lisp operations do not compile as of Version 6.1. (This list supersedes all such lists included in the **Lisp Release Notes* for previous versions.)

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

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

Parallel sequence operations do not compile. These operations are listed below.

copy–seq!!
***fill**
length!!
***nreverse**
subseq!!

every!!	**notany!!**	**notevery!!**	**some!!**

count!!	**count–if!!**	**count–if–not!!**
find!!	**find–if!!**	**find–if–not!!**
nsubstitute!!	**nsubstitute–if!!**	**nsubstitute–if–not!!**
position!!	**position–if!!**	**position–if–not!!**
substitute!!	**substitute–if!!**	**substitute–if–not!!**

The one exception is the **reduce!!** operation, which compiles in limited cases (see Section 6.2.2, below).

6.2.2 *Lisp Compiler Restrictions

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 had the value 2^{32} then **ash!!** would try to create a pvar roughly 2^{32} bits in length!)

Declarations that explicitly specify the length of the *count–pvar* argument will compile. For example, **(ash!! (the (unsigned–byte 4) pvar) (the (field–pvar 4) x–pvar))** will compile because the result can at most be 19 bits in length (4 bits from the source **pvar**, shifted by up to 15 bits as specified by **x–pvar**).

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.

6.3 Special Forms That Compile

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

compiler–let	**let**	**let***	**progn**
multiple–value–bind	**multiple–value–let**		

All other Common Lisp special forms, in particular **if** and similar conditionals, require explicit declaration of their returned value in order to be compiled by the *Lisp compiler.

6.4 Type Declarations and the *Lisp Code Walker

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

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.

6.4.1 The Code Walker

The *Lisp compiler includes a code walker that allows it 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.
- Compile 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.

6.4.2 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**.

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

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

6.4.3 Enabling and Disabling 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:

- 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.

- 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.

6.4.4 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
                             ;;;  are no longer needed
```

which will also compile into Paris code.

6.5 *Lisp Compiler Implementation Errors Update

All previously reported *Lisp compiler implementation errors have been corrected for the release of *Lisp Version 6.1.

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.

6.5.1 Known Errors Still Open

ID long–progn–compiler–stack–problem

Environment

*Lisp Version 6.1; Symbolics Genera 8.1; any CM configuration.

Synopsis

Compiling long forms with the code walker causes a stack error.

Description

Forms that allow any number of body forms are compiled by the *Lisp code walker using stack recursion. A large enough number of body forms can exhaust the stack, triggering an error on the Symbolics machine.

Reproduce By

```
(defun foo ()
     (+!! 2 3) ... <repeated 200-300 times> )

(compile 'foo)

Error: the control stack overflowed.
...
```

Workaround

Enclose the body forms into a number of smaller **progn** forms. Each is handled as a separate statement, reducing the total stack space required to compile the entire form.

Status

Open.

6.6 Miscellaneous Compilation Notes

6.6.1 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.

6.6.2 *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 6.1

NOTE: Version 6.1 of *Lisp introduces a new version, F19, of the *Lisp simulator.

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 *Lisp Simulator Is Freely Available

The *Lisp simulator is 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–f19–sharfile**. 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 sites that do not have a Connection Machine.

Thinking Machines Corporation will for the most part provide *Lisp simulator support only to 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.

7.2 *Lisp Simulator Restrictions Update

All 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.

7.2.1 Known Simulator Restrictions

7.2.1.1 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.

7.2.1.2 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)
```

7.2.2 Known Implementation Errors

All known simulator implementation errors are reported here, in alphabetical order by bug report ID.

ID lucid–exit–from–sim–bug

Environment

*Lisp simulator Version F18, F19; 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.

7.2.3 Notes on Simulator Use

7.2.3.1 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.

7.2.3.2 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–F19**, that may be used to conditionalize code that should be executed only in the F19 version of the *Lisp simulator. This symbol may be used in a manner similar to those shown above.

8 Sun and Lucid Common Lisp

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

*Lisp on a Sun-4 front end requires Sun Common Lisp 4.0. This is new as of Version 6.1. *Lisp on a VAX front end requires Lucid Common Lisp 2.5. This is unchanged from Version 6.0.

8.1 Differences between Sun and Lucid Common Lisp

There are significant differences between Sun Common Lisp and Lucid Common Lisp. The differences that most affect *Lisp programs are noted below.

Name changes:

The package name for system functions differs between Lucid and Sun Common Lisp. Functions in the **SYS** package in Lucid Lisp belong to the **LCL** package in Sun Lisp. For example, **(SYS::quit)** in Lucid Lisp corresponds with **(LCL::quit)** in Sun Lisp. (To write code that is portable between the two versions, you can use the package name **LUCID**. For example, **(LUCID::quit)** will execute in both Lucid 2.5 and Sun 4.0.)

The change–memory–management function:

Use of the **change–memory–management** function is not recommended. Only programs that contain large amounts of code, or require heavy garbage collection, should require you to use this function.

Lucid Common Lisp users should ask their site manager or applications engineer for assistance in determining the proper arguments to supply to this function.

For Sun Common Lisp users, if this function is used, it should be called with the following arguments 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.

Thereafter, if running *Lisp code causes excessive garbage collection, typing the following form may help:

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

The output produced by typing **(room t)** includes information about the current memory management settings.

Two Lucid compiler modes: Production and development

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.

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 4.0 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 4.0 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 4.0 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.

8.2 The *Lisp Compiler and the Common Lisp Compiler

The *Lisp compiler is completely independent of the Sun/Lucid Common 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.

For more information on the *Lisp compiler, refer to the overview chapter in *Getting Started in *Lisp* and the chapter on compiler options in the **Lisp Dictionary*. The **Lisp Compiler Guide* also contains useful information.

For more information on the Sun/Lucid Lisp compiler, refer to the system documentation for each Lisp environment.

8.3 Common Lisp Implementation Errors and Restrictions

All known outstanding Sun/Lucid-related implementation errors and restrictions are reported in these release notes. All past issues of Programming in *Lisp *In Parallel* and all previous **Lisp Release Notes* may therefore be discarded.

The following previously reported problems have been fixed in Version 6.1:

load–n–defstruct–wrong–warning
lucid–floating–point–compiler–bug

8.3.1 Known Errors Still Open

ID lucid–byte–specifier–size–limit

Environment

Lucid Common Lisp; Sun Common Lisp.

Synopsis

Both Lucid and Sun Common Lisp impose 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, posi-
tion=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.

ID lucid–describe–bus–error–on–pvar

Environment

Lucid Common Lisp, Sun Common Lisp.

Description

Calling **describe** on a pvar signals a bus error because Lucid can't handle the pvar data structure.

Reproduce By

```
(describe (!! 3.0))
```

Workaround

Use **describe–pvar** instead.

Status

Open.

ID non–standard–file–load–problem

Environment

*Lisp, Version 6.0; any CM configuration;
Lucid Common Lisp and Sun Common Lisp.

Synopsis

Non-standard extension Lisp code files don't load properly.

Description

The **load** function recognizes a special set of "standard" extensions for Lisp code files ("**.lisp**","**.bin**", etc.), and, if called on a file with a non-standard extension, **load** attempts to load the file as if it is a "**.bin**" file.

Reproduce By

```
> (load "test.data")
;;; Loading binary file "test.data"
>>Error: Invalid or garbled fasload (binary) file.

FASLOAD:
   Required arg 0 (FILENAME): #P"/test.data"
   ...
```

Workaround

Add the non-standard extension to the list **tmc:*load–source–pathname–types***, as in:

```
(push "data" tmc:*load-source-pathname-types*)
```

Status

Open.

9 *Lisp Library 6.1

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.

9.1 *Lisp Library Contents

The following interface files are included in the *Lisp library in Version 6.1:

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

9.2 *Lisp Library Restrictions Update

All previously reported *Lisp library implementation errors and restrictions have been corrected in the release of *Lisp Version 6.1.

9.3 Accessing the *Lisp Library

The *Lisp library code is available in the directory

```
/cm/starlisp/library/f6100/*
```

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

```
/cm/starlisp/library/f6100/documentation.text
```

Ask your system 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, all of the functions in that interface file are autoloaded.

10 *Graphics Version 6.1

*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 latest **Graphics Release Notes* (September 1990, as of this writing).

10.1 Improvements in Version 6.1

Faster Image Transfer. *Graphics users will notice faster transfer of images to displays thanks to improvements of the low-level graphics code.

11 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 Thinking Machines 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.