The
Connection Machine
System

*Lisp Release Notes

Version 5.0
September 1988

These release notes
replace all previous
*Lisp release notes

Thinking Machines Corporation
Cambridge, Massachusetts

First Printing, September 1988

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

Contents

Customer Support

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

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

To contact Thinking Machines Customer Support:

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

For Symbolics users only:

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

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

Please supplement the automatic report with any further pertinent information.

1 About Version 5.0

*Lisp is an extension of Common Lisp for data parallel programming on the Connection Machine. Programs using *Lisp typically include both Common Lisp and *Lisp constructs.

Version 5.0 is a major *Lisp release; it includes substantial additions and enhancements to previous versions. *Lisp Version 5.0 offers significant improvements in performance along with new programming features, enhanced error checking, and numerous corrections of previous implementation errors.

With this version, two significant new language features become available: *n-dimensional NEWS communications* and *virtual processor sets*. These new capabilities allow the address space for Connection Machine processors to be defined as multiple, *n*-dimensional coordinate systems. Thus, many limitations previously placed on interprocessor communication and on the use of virtual processors have been lifted. Also new in this version is support for numerous new pvars types including complex number pvars, array pvars, and user-defined structure pvars. Furthermore, experimental parallel vector and parallel sequence operations are provided with *Lisp Version 5.0.

More completely than previous versions, this version of *Lisp harnesses the power and versatility of the new Connection Machine hardware, the CM-2. For a detailed description of the relationship between *Lisp, Paris, and the CM-2, see Appendix A of the *Supplement to the *Lisp Reference Manual.*

1.1 Components of *Lisp, Version 5.0

Thinking Machines Corporation's implementation of *Lisp includes:

- The *Lisp interpreter, which executes *Lisp code on the full Connection Machine system
- The *Lisp compiler, which translates *Lisp code into Lisp/Paris for execution on the full Connection Machine system (or on the Paris simulator)
- The *Lisp simulator, which executes *Lisp code on a serial front-end computer alone, simulating the semantics of *Lisp

The *Lisp interpreter, simulator, and compiler can be used from within a Common Lisp environment on any Connection Machine (CM) front end. The CM front ends currently supported are the Symbolics Lisp machine, the VAX running the ULTRIX system, and the Sun-4 workstation running UNIX.

1.2 *Lisp Documentation

The documentation currently provided with *Lisp is listed below in the recommended reading sequence.

- The **Lisp Reference Manual*, Version 5.0, revised October 1988
- The **Lisp Release Notes*, Version 5.0, 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*, Version 5.0, is the primary source of information on the essential concepts and constructs of the *Lisp language. It is a revision of the **Lisp Reference Manual*, Version 4.0, updated to correct information presented in that publication and to account for the implementation *Lisp Version 5.0.

The **Lisp Release Notes*, Version 5.0, describe the enhancements, corrections, and restrictions to *Lisp since the publication of the **Lisp Reference Manual*, Version 4.0. Detailed information about *Lisp features new with this release is presented in the *Supplement to the *Lisp Reference Manual*, Version 5.0, and in the **Lisp Compiler Guide*, Version 5.0.

NOTE

Earlier *Lisp release notes—for V4.0, V4.1, V4.2, and V4.3—should be discarded. Earlier information that is still relevant to Version 5.0 is reproduced either in the present release notes or in the *Supplement to the *Lisp Reference Manual.*

The **Lisp Reference Manual*, Version 4.0 should be discarded; it is replaced by the **Lisp Reference Manual*, Version 5.0.

The **Lisp Compiler Guide*, Version 4.2A Field Test, should be discarded; it is replaced by the **Lisp Compiler Guide*, Version 5.0.

1.3 Organization of Release Notes

These release notes are divided into six sections as described below.

1 **About Version 5.0**
This section introduces the major new features of this version and describes the documentation provided with *Lisp, Version 5.0.

2 **Porting Code to Version 5.0**
The next section contains instructions for porting existing code to Version 5.0.

3 **Enhancements to the *Lisp Language**
The third section provides an overview of new *Lisp features, documented for the first time with Version 5.0.

4 **The *Lisp Interpreter**
The fourth section explains that the *Lisp interpreter has been completely reimplemented for the release of Version 5.0. Remaining restrictions are described.

5 **The *Lisp Compiler**
The fifth section focuses on the compiler, describing the enhancements and corrections made for the current version. Current restrictions are also detailed.

6 **The *Lisp Simulator**
The sixth section describes the current version of the *Lisp simulator.

2 Porting Code to Version 5.0

*Lisp Version 5.0 requires Connection Machine System Software Version 5.0. *Lisp programs compiled under previous versions should be recompiled to run under Version 5.0. That is, *Lisp programs compiled under versions V4.0, V4.0-1, V4.1, V4.2, and V4.3 should be recompiled to run on Version 5.0.

*Lisp Version 5.0 is not entirely backward-compatible with previous versions. The few incompatibilities are listed below along with instructions for modifying existing *Lisp code to run under Version 5.0. Before recompiling existing code, read this section completely and make all the necessary modifications.

2.1 Syntax of *pset Changed

The arguments to ***pset** have changed. The new syntax for ***pset** is:

***pset** *combiner value-pvar dest-pvar cube-address-pvar* [*Macro*]
&key :notify :collision-mode :vp-set

The required arguments have not changed from previous versions. The optional *notify* and *collision-mode* arguments to ***pset** now require keyword specification.The **:vp-set** keyword argument has been added.

Code written according to the syntax of earlier versions will, in most cases, still work under Version 5.0. If this is attempted, however, a warning is issued at compile-time. It is planned that future versions of *Lisp will signal an error if the new syntax is not used. Therefore, users are strongly encouraged to modify existing code to conform to the new syntax.

Existing code that uses ***pset** forms with the old *notify-pvar* and *collision-mode* optional arguments should be modified to run under Version 5.0. The values of these optional arguments need not be changed, but they should be preceded by the **:notify** and **:collision-mode** keywords, respectively.

The new **:vp-set** keyword argument may be used to specify the VP set to which *dest-pvar* belongs.

The concept of VP sets is explained in chapter 7 of the *Supplement to the *Lisp Reference Manual*. A detailed description of the ***pset** operation appears in chapter 8 of that volume.

2.2 Arguments to and Semantics of pref!! Changed

The arguments to pref!! have changed. The new syntax for pref!! is:

pref!! *pvar-expression cube-address-pvar* [*Macro*]
&key :collision-mode :vp-set

The required arguments have not changed from previous versions. The optional *collision-mode* argument to **pref!!** now requires keyword specification, **:collision-mode**, and the default *collision-mode* value has changed. The **:vp-set** keyword argument has been added.

Code written according to the syntax of earlier versions will, in most cases, still work under Version 5.0. If this is attempted, however, a warning is issued at compile-time. It is planned that future versions of *Lisp will signal an error if the new syntax is not used. Therefore, users are strongly encouraged to modify existing code to conform to the new syntax.

The default *collision-mode* value has changed from **:collisions-allowed** to **nil**. Existing code that specified no *collision-mode* argument and thereby relied on the old *collision-mode* default, will now get the new default. Due to this change in the semantics of **pref!!**, *collision-mode* specification should be reconsidered.

The new default, **:collision-mode nil**, invokes a Paris instruction (**cm:get-IL**), which uses the CM-2 backward routing hardware. As the number of collisions increases, this tends to be faster than **:collisions-allowed** and **:many-collisions**, but it uses much more temporary memory.

If a **pref!!** form causes *no* collisions, specify **:collision-mode** as **:no-collisions**. If there are *few* collisions, specify **:collisions-allowed** or use the default, **nil**. If there are *many* collisions, specify **:many-collisions** or use the default, **nil**. These last choices must be made heuristically. While the *collision-mode* default, **nil**, is faster than **:collisions-allowed** or **:many-collisions** when there are many collisions, it uses more memory. Try using the default. If the program runs out of memory, change the *collision-mode* for the offending **pref!!** form(s) to **:many-collisions** or **:collisions-allowed**.

The new **:vp-set** keyword argument is used to specify the VP set to which *pvar-expression* belongs. Specifying *vp-set* is mandatory if *pvar-expression* is an expression and if this expression should be evaluated in a VP set other than the current VP set. Otherwise, *vp-set* is optional.

The concept of VP sets is explained in chapter 7 of the *Supplement to the *Lisp Reference Manual*. A detailed description of the **pref!!** operation appears in chapter 8 of that volume.

2.3 Obsolete Language Features

An obsolete *Lisp language feature is one that should no longer appear in *Lisp code. If used, features rendered obsolete by Version 5.0 will generally signal a warning at compile-time. In the next release (probably Version 5.1) use of any one of these features will cause an error to be signaled at compile-time. It is therefore strongly recommended that existing *Lisp code using any of the following obsolete features be rewritten to use new, alternative constructs.

scan-grid!!	**pref-grid**
pref-grid!!	**pref-grid-relative!!**
***pset-grid**	***pset-grid-relative**
with-*lisp-from-paris	**with-paris-from-*lisp**
(setf (pref!! ...))	**(setf (pref ...))**

The discussion below covers how to replace each of these features.

2.3.1 Communication Operations

With the introduction of *n*-dimensional NEWS communications, the following constructs are rendered obsolete.

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

pref-grid *pvar* **&rest** *indices* [*Macro*]
pref-grid!! *pvar x-pvar y-pvar* **&key :border-pvar :collision-mode** [*Macro*]
pref-grid-relative!! *pvar* **&rest** *index-pvars* **&key :border-pvar** [*Macro*]

***pset-grid** *combiner value-pvar dest-pvar x-pvar y-pvar* [*Macro*]
&optional *notify-pvar collision-mode*
***pset-grid-relative** *combiner source-pvar dest-pvar* **&rest** *index-pvars* [*Macro*]

Failure to convert existing code to use new *Lisp constructs can result in significant loss of performance, both interpreted and compiled. This is especially true in the case of **pref-grid-relative!!**, which should be replaced by **news!!** or **news-border!!**.

For examples showing how to rewrite existing code that uses the above constructs, see the last few pages of chapter 8 in the *Supplement to the *Lisp Reference Manual.*

2.3.2 *Lisp-Paris Transition Macros

*Lisp Version 5.0, renders the following macro forms obsolete.

with-*lisp-from-paris (&body *body*) [*Macro*]
with-paris-from-*lisp (&body *body*) [*Macro*]

In previous versions, the use of these forms was required prior to calling Paris from *Lisp and prior to calling *Lisp from Paris, respectively.

*Lisp used to maintain a stack discipline different from the Paris stack discipline. With Version 5.0, *Lisp uses the Paris stack directly. Therefore, whereas these operations were previously needed to set up the appropriate stack discipline for the language in use, this is no longer necessary.

2.3.3 *proclaim Replaces proclaim in *Lisp Code

To make global pvar type declarations, ***proclaim** should now be used instead of **proclaim**. Previous releases of *Lisp redefined the Common Lisp **proclaim** construct to allow both the *Lisp interpreter and compiler to attend to **proclaim** forms. With the release of Version 5.0, this redefinition is being phased out. In future versions, *Lisp will not attend to **proclaim** forms at all; the only way to make global type declarations

and to set *Lisp compiler options will be by using ***proclaim**. It is strongly advised, therefore, that all uses of **proclaim** be replaced by ***proclaim** in *Lisp code.

2.3.4 *setf Replaces setf in *Lisp Code

Previous releases of *Lisp redefined the Common Lisp setf construct to allow both the *Lisp interpreter and compiler to attend to **setf** forms. With the release of Version 5.0, this redefinition is being phased out. In future versions, *Lisp will not attend to **setf** forms at all. It is strongly advised, therefore, that all uses of **setf** be replaced by ***setf** in *Lisp code. Note that this does *not* apply to uses of **setf** in Common Lisp forms.

2.4 Meaning of the float-pvar Type Specifier Changed

Whereas the type specifier **float-pvar** was, in previous versions, defined as equivalent to the type specifier **single-float-pvar**, it is now defined as equivalent to **(pvar float)**. More specifically, **float-pvar** used to be equivalent to **(pvar (defined-float 23 8))**, it is now equivalent to **(pvar (defined-float * *)**. This change brings the names of floating-point type specifiers into congruence with those of other pvar types. Existing code that relies on the old meaning of the **float-pvar** type specifier should substitute the **single-float-pvar** specifier for the **float-pvar** specifier throughout. For a detailed description of *Lisp type specifiers, see chapter 1 of the *Supplement to the *Lisp Reference Manual.*

2.5 How to Force Version-Specific Execution

It is possible to have both Version 5.0 and Version 4.3 available on CM front-end computers. For instructions on setting up such an environment, see the *System Front End Release Notes, V5.0* in the volume entitled *Connection Machine Front End Subsystems.* If both Version 5.0 and Version 4.3 are in use, it may be necessary to conditionalize code for appropriate execution under each version.

To force the Lisp reader to conditionally read a form depending on which version of the Connection Machine System Software (including *Lisp) is loaded, use the Common Lisp **#+** and **#-** reader macros with the feature symbol **CM-5.0**. Thus,

#+CM-5.0 *form*

reads *form* only if Version 5.0 of the Connection Machine System Software is loaded.

#-CM-5.0 *form*

reads *form* only if Version 5.0 of the Connection Machine System Software is *not* loaded.

Inserting these switches in existing code allows line-specific modifications that accommodate the few syntax changes introduced with Version 5.0 while allowing that code to continue to execute under Version 4.3.

Examples:

```
#+CM-5.0 (*pset :max from-pvar to-pvar address-pvar
           :notify whogot-pvar :collision-mode :many-collisions
           :vp-set to-vp)
#-CM-5.0 (*pset :max from-pvar to-pvar address-pvar
           whogot-pvar :many-collisions)
```

In the code above, a ***pset** form with keyword arguments, including a destination VP set, is read if Version 5.0 is loaded and a ***pset** form without keyword arguments is read if an earlier version is loaded.

```
(*set destination
#+CM-5.0 (pref!! source-pvar address-pvar
           :collision-mode :no-collisions :vp-set from-vp)
#-CM-5.0 (pref!! source-pvar address-pvar :no-collisions)
)
```

Here, a **pref!!** form with keyword arguments, including a source VP set, is read if Version 5.0 is loaded and a **pref!!** form without keyword arguments is read if an earlier version is loaded.

NOTE

Code using such reader macros must be recompiled when moving between versions.

3 Enhancements to the *Lisp Language

This section provides a brief overview of all the changes and additions made to the *Lisp language with Version 5.0. It does not attempt to describe these in detail. Formal definitions of all new language features are given in the *Supplement to the *Lisp Reference Manual*, Version 5.0. The supplement also contains code examples and programming hints helpful in making proper use of the new features.

The following major enhancements to the *Lisp language are introduced with *Lisp Version 5.0:

- Support for multiple, *n*-dimensional virtual processor sets (VP sets)
- Support for *n*-dimensional interprocessor communication (*N*-D NEWS)
- Support for the following new types of parallel variables (pvars):
  - array pvars
  - structure pvars
  - complex number pvars
  - character pvars
- Support for more flexible scan operations using *segment sets*
- Experimental parallel vector operations
- A complete set of efficient trigonometric and transcendental function for both floating point pvars and complex number pvars.

3.1 *N*-Dimensional NEWS and Virtual Processor Sets

Previous versions of *Lisp restricted the assignment of Connection Machine virtual processors to a two-dimensional grid pattern. This assignment could only be made once during a session: at the start, with the ***cold-boot** command. When the processor requirements of program data varied, users were forced to use the maximum number of virtual processors required by any data a program used. When smaller data were processed, portions of the machine had to be left idle.

*Lisp Version 5.0 introduces support for an *n*-dimensional configuration of Connection Machine processors, where *n* is any positive integer. The definition of an *n*-dimensional processor pattern results in an object known as a *virtual processor set* (or *VP set*). Multiple VP sets may be defined, each with separate data. Numerous new operations are provided to create, deallocate, and assign pvars to virtual processor sets.

The introduction of VP sets enables more efficient utilization of the Connection Machine system. First, assigning data to appropriately-dimensioned virtual processor sets avoids the problem of idle processors. Second, the implementation of virtual processor memory management has been improved with this version of the Connection Machine System Software.

Near neighbor communication between processors, termed *NEWS* communication, has been extended to *n*-dimensions. Also, several new operations are provided to support communication between virtual processor sets.

A detailed discussion of virtual processor sets and formal definitions of VP set operations may be found in the chapter entitled "Virtual Processor Sets" in the *Supplement to the *Lisp Reference Manual*, Version 5.0

A detailed discussion of *N*-D NEWS and formal definitions of *N*-D NEWS operations may be found in the chapter entitled "*N*-Dimensional Interprocessor Communication" in the *Supplement to the *Lisp Reference Manual*, Version 5.0.

3.2 New Pvar Types

*Lisp Version 5.0 adds four new parallel variables types to *Lisp: array pvars, structure pvars, complex pvars, and character pvars. The introduction of these data types adds flexibility to data representation on the CM and allows front-end data to be more easily broadcast to the CM processors.

3.2.1 Array Pvars

Prior to the release of Version 5.0, array pvars were not implemented. *Lisp users could create arrays of pvars on the front end and thus simulate the array pvar data structure. This is no longer necessary.

*Lisp defines array pvars as the parallel equivalent of Common Lisp arrays: one array is stored in each processor. Numerous functions are provided to create and manipu-

late array pvars. The new *Lisp operations specific to array pvars fall into the following categories:

- Creating and copying array pvars
- Describing and testing array pvars
- Storing and accessing array pvar contents
- Mapping functions over array pvars
- Performing logical bitwise operations on array pvars with single bit elements
- Fast indirect addressing using sideways arrays

For a detailed discussion of array pvars and for formal definitions of operations on array pvars, see the chapter entitled "Array Pvars," in the *Supplement to the *Lisp Reference Manual*, Version 5.0.

3.2.2 Structure Pvars

Prior to the release of Version 5.0, an experimental version of structure pvars was implemented. Structure pvars have been reimplemented with substantial changes in syntax and semantics.

*Lisp defines structure pvars as the parallel equivalent of Common Lisp structures. A structure pvar is a user-defined aggregate parallel data type. Evaluation of a pvar structure definition creates constructor, accessor, and assignment operations for that structure pvar. In addition, a structure pvar definition defines a front-end scalar structure and front-end operations on that structure. Creating a structure pvar means storing one instance of a defined structure in each active processor.

Several operations are provided to create and manipulate structure pvars. The following categories of new *Lisp operations specific to structure pvars have been implemented for Version 5.0:

- Defining and creating structure pvars
- Testing structure pvars
- Storing and accessing data in structure pvars

Inheritance among *Lisp structure pvar definitions may be employed to any depth. However, inheritance is limited to one ancestor per definition.

For a detailed discussion of structure pvars and for formal definitions of operations on structure pvars, see the chapter entitled "Structure Pvars," in the *Supplement to the *Lisp Reference Manual*, Version 5.0.

3.2.3 Complex Pvars

Prior to the release of Version 5.0, pvars could not contain complex numbers.

*Lisp defines complex pvars as the parallel equivalent of Common Lisp complex numbers: one complex number is stored in each active processor.

Parallel equivalents of all Common Lisp operations on complex numbers are provided with Version 5.0. Some functions that can return complex pvars bear argument restrictions more stringent than those placed on their Common Lisp analogues.

For a detailed discussion of complex pvars and for formal definitions of operations on complex pvars, see the chapter entitled "Complex Number Pvars," in the *Supplement to the *Lisp Reference Manual*, Version 5.0.

3.2.4 Character Pvars

Prior to the release of Version 5.0, pvars could not contain characters.

*Lisp defines character pvars as the parallel equivalent of Common Lisp characters: one character is stored in each active processor. Both the character pvar type and its string-char sub-type are supported by *Lisp.

Parallel equivalents of almost all Common Lisp operations on characters are provided with Version 5.0.

For a detailed discussion of character pvars and for formal definitions of operations on character pvars, see the chapter entitled "Character Pvars," in the *Supplement to the *Lisp Reference Manual*, Version 5.0.

3.3 Experimental Features

An *experimental feature* is one for which neither the design nor the implementation can be considered stable. Be aware that these features are less thoroughly tested than non-experimental features and are therefore more error-prone. Customers using these fa-

cilities are encouraged to offer suggestions and criticism. If well received, experimental features can become standard features. Otherwise, they may be removed from the language.

*Lisp Version 5.0 offers the following experimental features:

- **segment sets** for segmented scans
- **parallel vector operations**
- **parallel sequence operations**
- **address objects** for *N*-D NEWS communications

An overview of each experimental feature is provided below.

3.3.1 Enhanced Scanning Operations

The *Lisp facilities for segmented scan operations are now enhanced by the introduction of *segment sets*. A segment set is a group of potentially non-adjacent sequences of Connection Machine processors. When segment sets are used, the designated processors are selected—independent of the currently selected set—for the duration of the scan operation. New *Lisp operations are provided to create segment sets and to use them in parallel scans.

For a detailed discussion of segmented scans using segment sets, see the chapter entitled "Scanning with Segment Sets" in the *Supplement to the *Lisp Reference Manual*, Version 5.0.

3.3.2 Parallel Vector Operations

Experimental, parallel versions of standard operations on vectors are provided with *Lisp, Version 5.0. For instance, vector addition and dot product may be performed on vectors pvars containing numeric data. Experimental, optimized versions of vector operations are provided for vector pvars containing single-precision floating-point numbers. Also provided are experimental, serial equivalents of *Lisp vector functions for which no Common Lisp analogues exist.

For a detailed discussion of *Lisp vector operations, see the chapter entitled "Parallel Vector Operations" in the *Supplement to the *Lisp Reference Manual*, Version 5.0.

3.3.3 Parallel Sequence Operations

*Lisp defines a parallel sequence as a pvar containing a one-dimensional array in each processor. Experimental parallel equivalents of a subset of Common Lisp sequence operations are provided in *Lisp, Version 5.0. The following general categories of parallel sequence operations are now available in *Lisp:

- simple operations such as element subselection, copying, and reversing
- modifying parallel sequences
- searching parallel sequences
- mapping predicates over sequences in each processor

More stringent restrictions are placed on *Lisp sequence pvars than on Common Lisp sequences:

- Sequence pvars are one dimensional array pvars; list pvars are not supported.
- Sequence pvars must be composed of sequences that are—in each active processor—of fixed, uniform size.

A variety of restrictions are also placed on the arguments to individual parallel sequence operations.

For a detailed discussion of *Lisp parallel sequence operations, see the chapter entitled "Parallel Sequence Operations" in the *Supplement to the *Lisp Reference Manual*, Version 5.0.

3.3.4 Address Objects

A new approach to grid addressing is introduced with a feature known as address objects. Address objects simplify and generalize grid addressing across VP sets.

*Lisp provides functions for creating and manipulating address objects. Address objects are structures defined with ***defstruct** and containing grid coordinates. They are easier to use than other methods of translating between addresses.

For a detailed discussion of address objects, see the chapter entitled "*N*-Dimensional Interprocessor Communication" in the *Supplement to the *Lisp Reference Manual*, Version 5.0.

4 The *Lisp Interpreter

4.1 Interpreter Enhancements

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.

The *Lisp interpreter has been completely reimplemented for Version 5.0. This has resulted in the following improvements.

- New language features are supported.

  All the new language features mentioned in these release notes and detailed in the *Supplement to the *Lisp Reference Manual*, Version 5.0, are supported by the *Lisp interpreter.

- Performance is improved.

  *Lisp now makes better use of the CM-2 capabilities, thus offering performance gains over previous versions. For a discussion of the relevant features, see Appendix A of the *Supplement to the *Lisp Reference Manual.*

- Error handling is improved.

  More consistent error handling and reporting is now provided by *Lisp. This enhancement affects execution of nearly every *Lisp construct and is treated in chapter 7 of the *Supplement to the *Lisp Reference Manual.*

- Interpreter safety is available.

  It is now possible to choose between different levels of interpreter safety. This debugging aid is discussed in chapter 7 of the *Supplement to the *Lisp Reference Manual.*

IMPORTANT

The efficient execution of interpreted codes is highly dependent upon the value of ***interpreter-safety***. The user is strongly advised to become familiar with the proper method of setting different safety levels.

4.2 Interpreter Restrictions

The following restrictions, which existed in *Lisp Version 4.3, still apply to *Lisp Version 5.0.

- 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** will 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.

5 The *Lisp Compiler

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

This section lists enhancements, corrections, and restrictions to the *Lisp compiler. The information in this section is new. It summarizes changes made to the compiler since publication of the **Lisp Compiler Guide,* Version 4.2A Field Test, in October 1987 and the **Lisp Release Notes*, Version 4.3, in January 1988. For a detailed description of the *Lisp compiler, see the newly-revised **Lisp Compiler Guide,* Version 5.0.

5.1 Compiler Enhancements

The *Lisp compiler, Version 5.0, contains the following enhancements.

- The compiler is now, by default, on.
- The compiler menu options have changed.
- The degree of type checking performed by the compiler is now configurable.
- Some *Lisp operations that did not previously compile now do.

Detailed descriptions of these enhancements may be found in the **Lisp Compiler Guide*, Version 5.0. A summary is given below.

5.1.1 Compiler Now ON by Default

Previously, the *Lisp Compiler had to be explicitly enabled from the options menu. The *Lisp compiler is now on by default. The *Lisp compiler can translate most but not all *Lisp statements into Lisp/Paris. Any *Lisp statement that cannot be translated is interpreted by the *Lisp interpreter.

5.1.2 Compiler Menu Option Changes

The following compiler options have been added to the full options menu.

Add Declares
Use Undocumented Paris
Verify Type Declarations
Constant Fold Expressions
Speed
Compilation Speed
Space
Immediate Error if Location
Optimize Check Stack Expression
Generate Comments with Paris Code

While the functionality of most of these options is not new, it is now possible to more finely adjust compiler operation by changing the value of one or more of these options. In most cases this is not necessary.

The default values for the following compiler options have changed.

Compile Expressions
Warning Level
Optimize Bindings
Peephole Optimize Paris
Machine Type

The **Compile Expressions** default was **No**, it is now **Yes**; the *Lisp compiler is now on by default.

The **Warning Level** default was **High**, it is now **Normal**. This means that, by default, the *Lisp compiler does not report failure every time it is unable to process a *Lisp statement that it attempts to *compile.

The **Optimize Bindings** and **Peephole Optimize Paris** compiler options both now default to **Cspeed<3**, a value which varies the degree of optimization based on the value of the compilation speed variable ***compilation-speed***.

The **Machine Type** compiler option default has been changed from **Compatible** to **Current**. This means that, by default, the compiler now generates code specific to the type of Connection Machine system currently in use. Previously it generated code compatible across Connection Machine systems.

The following compiler options have been removed from the default menu. While these options are still available on the complete menu, they seldom need to be given settings other than their defaults.

Optimize Bindings
Peephole Optimize
Machine Type

The **Machine Type** option value of CM1 no longer works; the *Lisp compiler will not generate code that works on the CM-1.

The **Use Paris Macros** *Lisp compiler option has been disabled; it is no longer available.

5.1.3 Configurable Compiler Type Checking

By setting the value of the new **Verify Type Declarations** compiler option, it is now possible to vary the amount of type checking performed by the compiler. By default, the level of type checking varies with the value of the compilation safety variable ***safety***.

5.1.4 *Lisp Operations that Compile

Most of the *Lisp operations introduced with Version 5.0 can be compiled. As with previous version of the *Lisp compiler, *Lisp expressions that *can* be compiled *are* compiled only if they are enclosed in forms that allow the compiler to "see" them. (See the **Lisp Compiler Guide*, Version 5.0, for information on which forms the compiler can "see.")

The following *Lisp operations that did not previously compile now do.

sin!! **cos!!** **log!!**

Several of the restrictions previously placed on the functions **load-byte!!** and **ldb!!** in order to allow compilation have now been relaxed. The syntactic formats for **load-byte!!** and **ldb!!** are:

load-byte!! *source-pvar position-pvar size-pvar* [*Function*]
ldb!! *bytespec-pvar source-pvar* [*Function*]

These two functions are equivalent in the following way.

```
(load-byte!! source-pvar position-pvar size-pvar)
<=>
(ldb!! (byte!! size-pvar position-pvar) source-pvar)
```

Previously, *source-pvar* had to be of a known integer length or the expression would not compile. The length of the *source-pvar* argument can now be indefinite, known only at run time.

The *position-pvar* argument no longer must be of the form (!! *integer-constant*). Now it may be either (!! *integer-expression*) or a pvar containing different values in each processor.

The *size-pvar* argument no longer must be of the form (!! *integer-constant*). It may now be of the form (!! *integer-expression*). However, *size-pvar* may not be a pvar containing different values in different processors.

Example:

```
(load-byte!! u8 (!! 5) (!! 3))

(load-byte!! u-cube u4 (!! j))
```

Here, **u8** is of type **(unsigned-byte 8)**, **u-cube** is and unsigned integer pvar whose length varies with the the cube address length, **u4** is of type **(unsigned-byte 4)**, and **j** is any integer. The first of the above expressions would previously compile whereas the second would not. Now both forms will compile.

5.2 Compiler Restrictions

The current version of the *Lisp compiler does not yet compile all *Lisp functions. Anything that is not compiled is handled by the interpreter. If the **Warning Level** compiler option is set to **High**, the compiler prints a warning whenever an operation is *visible* but not compiled. (See the **Lisp Compiler Guide* for the definition of operations visible to the *Lisp compiler.)

5.2.1 Forms Not Yet Compiled

Arguments to user-defined functions are currently not compiled. In addition, the *Lisp operations listed below are currently not compiled by the *Lisp compiler.

char-bit!!	**set-char-bit!!**
digit-char!!	

These operations on character pvars are described in chapter 2 of the *Supplement to the *Lisp Reference Manual.*

array-in-bounds-p!!	**make-array!!**
array-rank!!	***array-rank**
array-dimensions!!	***array-dimensions**
array-dimension!!	***array-element-type**
***array-total-size**	**array-total-size!!**
array-row-major-index!!	**typed-vector!!**
***sideways-array**	**sideways-aref!!**
***map**	

These operations on parallel arrays are described in chapter 3 of the *Supplement to the *Lisp Reference Manual.*

(!! *CL-structure***)**	**(*setf (pref** *parallel-structure* **)**
parallel-structure-**p!!**	**structurep!!**

Where *CL-structure* is a Common Lisp structure and *parallel-structure* is defined by ***defstruct**. Operations on parallel structures are described in chapter 4 of the *Supplement to the *Lisp Reference Manual.*

grid!!	**grid-relative!!**
***news**	**news-border!!**
address-nth!!	**address-plus!!**
address-pius-nth!!	**address-rank!!**
array-to-pvar	**array-to-pvar-grid**
pvar-to-array	**pvar-to-array-grid**
***pset-grid-relative**	

These addressing and communications operations are described in chapter 6 of the *Supplement to the *Lisp Reference Manual.*

byte!!	**ppp-address-object**
byte-size!!	**byte-position!!**
rot!!	**sort!!**
equalp!!	**dpb!!**
ppp!!	**pppdbg**

These miscellaneous operations are described in chapter 7 of the *Supplement to the *Lisp Reference Manual.*

In addition, none of the experimental parallel sequence operations and none of the exprimental parallel vector operations can be compiled. These new, experimental operations are described in chapters 10 and 11 of the *Supplement to the *Lisp Reference Manual.*

5.2.2 Forms Compiled with Restrictions

The following functions compile only if the restrictions noted below are observed.

aref!!

The *subsript-pvar* arguments hold different values from each other, but within each *subscript-pvar* argument identical values must be stored across all processors. That is, each must be of the form (!! *integer*).

pref

The *pvar-expression* argument may not be a parallel array of arrays nor a parallel array of structures.

pref!!
pref-grid!!

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

pref-grid-relative!!

The arguments *relative-address-pvars* must be constants such as (!! x) where x may be a variable or an integer.

***pset**
***pset-grid**

The *combiner* argument may not have the value **:default**, and the optional *collision-mode* argument may not have the value **:many-collisions**.

load-byte!!

The *size-pvar* argument must be constants such as (!! x), where x must be an integer.

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

Only the predicate is compiled; the body is not.

***let**
let

Only the initial values are compiled; the body is not.

deposit-byte!!

The *value* and *into-value* arguments must be unsigned-byte pvars of definite length. The *position* and *size* arguments must be textually !! forms.

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

If the **&optional** *bit-array-result-pvar* is provided, none of these functions compile.

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

The result may not be used as the source pvar in a ***set** form.

5.2.3 Restrictions on Compiler Options

Several *Lisp compiler options have attenuated functionality in the current version.

Pull Out Common Address Expressions

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

Compilation Speed

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

Space

The compiler does not consider this option currently.

Use Always Instructions

This option is not fully implemented and may generate undocumented Paris instructions.

Speed

The execution speed level setting is not used.

5.3 Notes on Compiler Use

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.

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.

6 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

In the past, the *Lisp simulator has 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.

6.1 Simulator Version

The Version 5.0 *Lisp simulator is not yet available. Instead, users wishing to simulate *Lisp on a computer not connected to a Connection Machine system must continue to use the *Lisp simulator, Version 4.3. It is anticipated that an updated version of the simulator will be made available with the next minor version release.

Most of the items in this section have been reported with previous releases.

6.2 Simulator Restrictions

The *Lisp simulator supports only general pvars; it does not support any of the other pvar types.

The *Lisp simulator still bears the Version 4.3 interpreter's restriction on naming user-defined *Lisp functions. Because such functions circumvent the Common Lisp package system, they must all have unique names. (Note that this restriction no longer applies to the *Lisp interpreter.)

Prior to Version 4.3, declarations of the form **(type (pvar ...))** were not accepted. Any type declaration with **pvar** as its first symbol is now accepted, although in certain circumstances the simulator may issue a warning.

The twenty-two operations added to the *Lisp interpreter and compiler for Version 4.3 are not yet available in the *Lisp simulator. A list of these operations appears below. For definitions, see chapter 7 of the *Supplement to the *Lisp Reference Manual.*

lognand!! *integer-pvar1 integer-pvar2*
lognor!! *integer-pvar1 integer-pvar2*
logandc1!! *integer-pvar1 integer-pvar2*
logandc2!! *integer-pvar1 integer-pvar2*
logorc1!! *integer-pvar1 integer-pvar2*
logorc2!! *integer-pvar1 integer-pvar2*
boole!! *op-pvar integer-pvar1 integer-pvar2*
logbitp!! *index-pvar integer-pvar*
logtest!! *integer-pvar1 integer-pvar2*
logcount!! *integer-pvar*
integer-length!! *integer-pvar*
gcd!! &rest *integer-pvars*
lcm!! *integer-pvar* **&rest** *integer-pvars*
mask-field!! *bytespec-pvar integer-pvar*
ffloor!! *number-pvar* **&optional** *divisor-pvar*
fceiling!! *number-pvar* **&optional** *divisor-pvar*
ftruncate!! *number-pvar* **&optional** *divisor-pvar*
fround!! *number-pvar* **&optional** *divisor-pvar*
scale-float!! *float-pvar integer-pvar*

float-sign!! *float-pvar1* **&optional** *float-pvar2*
integer-from-gray-code!! *integer-pvar*
gray-code-from-integer!! *integer-pvar*

6.3 Simulator Correction: a Patch to *pset

To correct an error in the simulator's implementation of ***pset**, a patch should be installed. This patch may be found in the file

```
/cm/starlisp/interpreter/f5005/starlisp-simulator-patch.lisp
```

Ask your systems administrator or your applications engineer to direct you to the location of this file at your installation.

A copy of this file is included below as a backup.

```
;;; REPLACE THE FUNCTION *PSET-1 IN THE FILE ADDRESSING.LISP
;;; WITH THIS VERSION OF THE FUNCTION.  THEN RECOMPILE THE
;;; *LISP SIMULATOR.

(defun *pset-1

         (function combinator source-pvar dest-pvar address-pvar
          &optional (notify-pvar nil)
                    (collision-mode :collisions-allowed))

  (pvar-check address-pvar)
  (pvar-check source-pvar)
  (pvar-check-lvalue dest-pvar function)
  (when notify-pvar (pvar-check-lvalue notify-pvar function))
  (when collision-mode
    (check-collision-mode collision-mode function))
  (when (or (not (keywordp combinator))
            (null (setq combinator
                    (get combinator 'pset-function))))
    (error "~S a valid combinator: ~S" function combinator))
  (if (or (null *pset-collision-array*)
          (not (eq (array-dimension *pset-collision-array* 0)
                   *number-of-processors-limit*)))
      (setq *pset-collision-array*
            (make-array *user-number-of-processors*
                        :element-type 'null)))
```

```
(when (eq dest-pvar notify-pvar)
  (error "-S dest-pvar and the notify-pvar are identical.
    This makes no sense." function))

;; If there is any overlap anywhere, make copies of the
;; destination and notify pvars, just in case.

(if (or (eq source-pvar dest-pvar)
        (eq source-pvar address-pvar)
        (eq source-pvar notify-pvar)
        (eq dest-pvar address-pvar)
        (eq address-pvar notify-pvar)
        )

    (*let ((temp-source-pvar source-pvar)
           (temp-dest-pvar dest-pvar)
           (temp-address-pvar address-pvar)
           (temp-notify-pvar (if notify-pvar notify-pvar nil!!))
           )

       (*all (*set temp-dest-pvar dest-pvar))
       (*all (if notify-pvar (*set temp-notify-pvar notify-pvar)))

       (*pset-internal
         combinator
         (pvar-array temp-source-pvar)
         (pvar-array temp-dest-pvar)
         (pvar-array temp-address-pvar)
         (if notify-pvar (pvar-array temp-notify-pvar) nil)
         *pset-collision-array*
         )

       (*all (*set dest-pvar temp-dest-pvar))
       (*all (if notify-pvar (*set notify-pvar temp-notify-pvar)))

       )

    (*pset-internal
         combinator
         (pvar-array source-pvar)
         (pvar-array dest-pvar)
         (pvar-array address-pvar)
         (if notify-pvar (pvar-array notify-pvar) nil)
         *pset-collision-array*
         )
  ))
```

6.4 Notes on Simulator Use

6.4.1 Conditional Simulator 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 with the *Lisp simulator. This is especially helpful in situations like the present one, where code intended for the simulator must conform to Version 4.3 while code intended to execute on the Connection Machine hardware will likely use new constructs introduced with Version 5.0.

To force the Lisp reader to conditionally read a form depending on whether or not the 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 examples that use similar reader macros, see section 2.5 of these release notes.

6.4.2 *proclaim

The *Lisp simulator requires that the operation ***proclaim** be used to make global declarations. This behavior is now more compatible with the *Lisp interpreter. Whereas previous versions of the interpreter have relied on a redefinition of the Common Lisp **proclaim** operation to allow *Lisp to attend to global declarations, with the release of Version 5.0, the *Lisp interpreter makes the use of **proclaim** forms in *Lisp statements obsolete.

7 Floating-Point Hardware Problem

The 32-bit floating-point accelerator is an option available with the CM-2. This section applies only to CM-2 machines with the floating-point accelerator option. Early version of these chips have a hardware error that causes a slight loss in precision in some floating-point multiply results. New versions of the chips do not have this problem. The affected *Lisp functions are:

***!!**	**/!!**	**sqrt!!**
truncate!!	**floor!!**	**ceiling!!**
round!!	**mod!!**	**rem!!**
sin!!	**cos!!**	**log!!**

This problem affects *Lisp statements running either interpreted or compiled. The nature of this problem is documented in detail in the *System Front Ends Release Notes*, Version 5.0, which are in the binder labeled *Connection Machine Front-End Subsystems*. The interested reader is urged to consult that document.

To avoid getting precision errors, use the **with-fpu** macro located in the **cmi** package.

cmi::with-fpu &key :mode :form [*Macro*]

The value of the **:form** argument may be any *Lisp form.This macro evaluates *form*. If **:mode** is **:off**, the floating-point accelerator is turned off for the duration of the evaluation of *form*. If **:mode** is **:on**, the floating-point accelerator is turned on for the duration of the evaluation of *form*. For example

```
(cmi::with-fpu :mode :off :form (*!! pvar1 pvar2))
```

turns off the floating-point accelerator while executing **(*!! pvar1 pvar2)**.

8 New Operations Not Yet Documented

What follows is simply a list of new *Lisp operations which—with Version 5.0—have yet to be fully documented.

unproclaim
ldb-test!!
deposit-field!!
ldb!!
dpb!!
byte!!
byte-size!!
byte-position!!
case!!
***case**
ecase!!
***ecase**
deallocate-vp-sets
sideways-array-p
***incf**
***decf**