1. Introduction

1.1 General Information

The Lisp Machine is a new computer system designed to provide a high-performance and economical implementation of the Lisp language. It is a personal computation system, which means that processors and main memories are not time-multiplexed: when using a Lisp Machine, you get your own processor and memory system for the duration of the session. It is designed this way to relieve the problems of running large Lisp programs on time-sharing systems. Everything on the Lisp Machine is written in Lisp, including all system programs; there is never any need to program in machine language. The system is highly interactive.

The Lisp Machine executes a new dialect of Lisp called Zetalisp, developed at the M.I.T. Artificial Intelligence Laboratory for use in artificial intelligence research and related fields. It was originally based on the Maclisp dialect, and attempts to maintain a good degree of compatibility with Maclisp, while also providing many improvements and new features. Maclisp, in turn, was based on Lisp 1.5.

Common Lisp is a Lisp dialect designed to standardize all the various Lisp systems derived from Maclisp. Zetalisp today is nearly a superset of Common Lisp, but there are a few important incompatibilities between them, in places where Common Lisp involves an incompatible change which is deemed to severe to impose on traditional Zetalisp users. There is a special mode which provides strict Common Lisp compatibility. See section 1.4, page 7 for more information.

This document is the reference manual for the Zetalisp language. This document is not a tutorial, and it sometimes refers to functions and concepts that are not explained until later in the manual. It is assumed that you have a basic working knowledge of some Lisp dialect; you will be able to figure out the rest of the language from this manual.

There are also facilities explained in this manual that are not really part of the Lisp language. Some of these are subroutine packages of general use, and others are tools used in writing programs. The Lisp Machine window system and the major utility programs are, or ought to be, documented in other manuals.

1.2 Structure of the Manual

The manual starts out with an explanation of the language. Chapter 2 explains the different primitive types of Lisp object and presents some basic *predicate* functions for testing types. Chapter 3 explains the process of evaluation, which is the heart of the Lisp language. Chapter 4 introduces the basic Lisp control structures.

The next several chapters explain the details of the various primitive data-types of the language and the functions that deal with them. Chapter 5 deals with conses and the higher-level structures that can be built out of them, such as trees, lists, association lists, and property lists. Chapter 6 deals with symbols, chapter 7 with the various kinds of numbers, and chapter 8 with arrays. Chapter 10 explains character strings, which are a special kind of array.

After this there are some chapters that explain more about functions, function-calling, and related matters. Chapter 11 presents all the kinds of functions in the language, explains function-specs, and tells how to manipulate definitions of functions. Chapters 12 and 13 discuss closures and stack-groups, two facilities useful for creating coroutines and other advanced control and access structures.

Next, a few lower-level issues are dealt with. Chapter 14 explains locatives, which are a kind of pointer to memory cells. Chapter 15 explains the "subprimitive" functions, which are primarily useful for implementation of the Lisp language itself and the Lisp Machine's operating system. Chapter 16 discusses areas, which give you control over storage allocation and locality of reference.

Chapter 17 discusses the Lisp compiler, which converts Lisp programs into "machine language" or "macrocode". Chapter 18 explains the Lisp macro facility, which allows users to write their own extensions to Lisp, extending both the interpreter and the compiler. The next two chapters go into detail about two such extensions, one that provides a powerful iteration control structure (chapter 19), and one that provides a powerful data structure facility (chapter 20).

Chapter 21 documents flavors, a language facility to provide generic functions using the paradigm used in Smalltalk and related languages, called "object-oriented programming" or "message passing". Flavors are widely used by the system programs of the Lisp Machine, as well as being available to the user as a language feature.

The next few chapters discuss I/O: chapter 22 explains I/O streams and character and line level operations; chapter 23 explains reading and printing symbolic expressions; chapter 24 explains naming of files; chapter 25 explains input and output to files. Chapter 26 describes the use of the Chaosnet.

Chapter 27 describes the *package* system, which allows many name spaces within a single Lisp environment. Chapter 28 documents the "system" facility that helps you create and maintain systems, which are programs that reside in many files.

Chapter 29 discusses the facilities for multiple processes and how to write programs that use concurrent computation. Chapter 30 explains how exceptional conditions (errors) can be handled by programs, handled by users, and debugged. Chapter 31 explains the instruction set of the Lisp Machine and tells you how to examine the output of the compiler. Chapter 32 documents some functions for querying the user, chapter 34 explains some functions for manipulating dates and times, and chapter 35 contains other miscellaneous functions and facilities.

1.3 Notational Conventions and Helpful Notes

There are several conventions of notation and various points that should be understood before reading the manual. This section explains those conventions.

The symbol '=>' is used to indicate evaluation in examples. Thus, when you see 'foo => nil', this means that "the result of evaluating foo is (or would have been) nil".

The symbol '==>' is used to indicate macro expansion in examples. This, when you see '(foo bar) ==> (aref bar 0)', this means that "the result of macro-expanding (foo bar) is (or would have been) (aref bar 0)".

A typical description of a Lisp function looks like this:

function-name *arg1 arg2* &optional *arg3* (*arg4 arg2*)

The function-name function adds together *arg1* and *arg2*, and then multiplies the result by *arg3*. If *arg3* is not provided, the multiplication isn't done. function-name then returns a list whose first element is this result and whose second element is *arg4*. Examples:

```
(function-name 3 4) => (7 4)
(function-name 1 2 2 'bar) => (6 bar)
```

Note the use of fonts (typefaces). The name of the function is in bold-face in the first line of the description, and the arguments are in italics. Within the text, printed representations of Lisp objects are in a different bold-face font, as in (+ foo 56), and argument references are italicized, as in *arg1* and *arg2*. A different, fixed-width font, as in `function-name`, is used for Lisp examples that are set off from the text. Other font conventions are that filenames are in bold-face, all upper case (as in SYS: SYS; SYSDCL LISP) while keys on the keyboard are in bold-face and capitalized (as in Help, Return and Meta).

'Car', 'cdr' and 'cons' are in bold-face when the actual Lisp objects are being mentioned, but in the normal text font when used as words.

The word '&optional' in the list of arguments tells you that all of the arguments past this point are optional. The default value can be specified explicitly, as with *arg4* whose default value is the result of evaluating the form (foo 3). If no default value is specified, it is the symbol nil. This syntax is used in lambda-lists in the language, which are explained in section 3.3, page 38. Argument lists may also contain '&rest' and '&key' to indicate rest and keyword arguments.

The descriptions of special forms and macros look like this:

do-three-times *form* *Special form*

This evaluates *form* three times and returns the result of the third evaluation.

with-foo-bound-to-nil *form...* *Macro*

This evaluates the *forms* with the symbol foo bound to nil. It expands as follows:

```
(with-foo-bound-to-nil
    form1
    form2 ...) ==>
(let ((foo nil))
    form1
    form2 ...)
```

Since special forms and macros are the mechanism by which the syntax of Lisp is extended, their descriptions must describe both their syntax and their semantics; functions follow a simple consistent set of rules, but each special form is idiosyncratic. The syntax is displayed on the first line of the description using the following conventions. Italicized words are names of parts of the form which are referred to in the descriptive text. They are not arguments, even though they resemble the italicized words in the first line of a function description. Parentheses ('(' and ')') stand for themselves. Square brackets ('[' and ']') indicate that what they enclose is optional. Ellipses ('...') indicate that the subform (italicized word or parenthesized list) that precedes them may be repeated any number of times (possibly no times at all). Curly brackets followed by ellipses ('{' and '}...') indicate that what they enclose may be repeated any number of times. Thus the first line of the description of a special form is a "template" for what an instance of that special form would look like, with the surrounding parentheses removed. The syntax of some special forms is sufficiently complicated that it does not fit comfortably into this style; the first line of the description of such a special form contains only the name, and the syntax is given by example in the body of the description.

The semantics of a special form includes not only what it "does for a living", but also which subforms are evaluated and what the returned value is. Usually this will be clarified with one or more examples.

A convention used by many special forms is that all of their subforms after the first few are described as '*body*...'. This means that the remaining subforms constitute the "body" of this special form; they are Lisp forms that are evaluated one after another in some environment established by the special form.

This ridiculous special form exhibits all of the syntactic features:

twiddle-frob [(*frob option*...)] {*parameter value*}... *Special form*

This twiddles the parameters of *frob*, which defaults to **default-frob** if not specified. Each *parameter* is the name of one of the adjustable parameters of a frob; each *value* is what value to set that parameter to. Any number of *parameter/value* pairs may be specified. If any *options* are specified, they are keywords that select which safety checks to override while twiddling the parameters. If neither *frob* nor any *options* are specified, the list of them may be omitted and the form may begin directly with the first *parameter* name.

frob and the *values* are evaluated; the *parameters* and *options* are syntactic keywords and not evaluated. The returned value is the frob whose parameters were adjusted. An error is signaled if any safety checks are violated.

Operations, the message-passing equivalent of ordinary Lisp's functions, are described in this style:

:operation-name *arg1 arg2* &optional *arg3* *Operation on* flavor-name
This is the documentation of the effect of performing operation :operation-name (or, sending a message named :operation-name), with arguments *arg1*, *arg2*, and *arg3*, on an instance of flavor flavor-name.

Descriptions of variables ("globally special" variables) look like this:

typical-variable *Variable*
The variable typical-variable has a typical value....

If the description says 'Constant' rather than 'Variable', it means that the value is never set by the system and should not be set by you. In some cases the value is an array or other structure whose *contents* may be changed by the system or by you.

Most numbers in this manual are decimal; octal numbers are labelled as such, using #o if they appear in examples. Currently the default radix for the Lisp Machine system is eight, but this will be changed in the near future. If you wish to change to base ten now, see the documentation on the variables *read-base* and *print-base* (page 517).

All uses of the phrase 'Lisp reader', unless further qualified, refer to the part of Lisp that reads characters from I/O streams (the read function), and not the person reading this manual.

There are several terms that are used widely in other references on Lisp, but are not used much in this document since they have become largely obsolete and misleading. For the benefit of those who may have seen them before, they are: 's-expression', which means a Lisp object; 'dotted pair', which means a cons; and 'atom', which means, roughly, symbols and numbers and sometimes other things, but not conses. The terms 'list' and 'tree' are defined in chapter 5, page 86.

The characters acute accent (') (also called "single quote") and semicolon (';') have special meanings when typed to Lisp; they are examples of what are called *macro characters*. Though the mechanism of macro characters is not of immediate interest to the new user, it is important to understand the effect of these two, which are used in the examples.

When the Lisp reader encounters a " ' ", it reads in the next Lisp object and encloses it in a quote special form. That is, 'foo-symbol turns into (quote foo-symbol), and '(cons 'a 'b) turns into (quote (cons (quote a) (quote b))). The reason for this is that quote would otherwise have to be typed in very frequently, and would look ugly.

The semicolon is used as a commenting character. When the Lisp reader sees one, the remainder of the line is discarded.

The character '/' is used for quoting strange characters so that they are not interpreted in their usual way by the Lisp reader, but rather are treated the way normal alphabetic characters are treated. So, for example, in order to give a '/' to the reader, you must type '//', the first '/' quoting the second one. When a character is preceded by a '/' it is said to be *escaped*. Escaping also turns off the effects of macro characters such as " ' " and ';'.

If you select Common Lisp syntax, escaping is done with '\' instead, and '/' has no special syntactic significance. The manual uses traditional syntax throughout, however.

The following characters also have special meanings and may not be used in symbols without escaping. These characters are explained in detail in the section on printed representation (section 23.3, page 516).

" Double-quote delimits character strings.

Sharp-sign introduces miscellaneous reader macros.

' Backquote is used to construct list structure.

, Comma is used in conjunction with backquote.

: Colon is the package prefix.

| Characters between pairs of vertical-bars are escaped.

⊗ Circle-cross lets you type in characters using their octal codes.

All Lisp code in this manual is written in lower case. In fact, the reader turns all symbols into upper case, and consequently everything prints out in upper case. You may write programs in whichever case you prefer.

You will see various symbols that have the colon (:) character in their names. The colon and the characters preceding it are not actually part of the symbol name, but in early stages of learning the system you can pretend that they are. Actually they are a package prefix. See chapter 27 for an explanation of packages and what package prefixes really do.

Symbols whose names start with si: are internal to the system. These functions and variables are documented here because they are things you sometimes need to know about. However, they are subject to change with little concern for compatibility for users.

Zetalisp is descended from Maclisp, and a good deal of effort was expended to try to allow Maclisp programs to run in Zetalisp. Throughout the manual, there are notes about differences between the dialects. For the new user, it is important to note that many functions herein exist solely for Maclisp compatibility; they should *not* be used in new programs. Such functions are clearly marked in the text.

The Lisp Machine character set is not quite the same as that used on I.T.S. nor on Multics; it is described in full detail in section 10.1.1, page 205. The important thing to note for now is that the character "newline" is the same as Return, and is represented by the number 215 octal. (This number should *not* be built into any programs.)

When the text speaks of "typing Control-Q" (for example), this means to hold down the Control key on the keyboard (either of the two keys labeled 'CTRL'), and, while holding it down, to strike the Q key. Similarly, to type Meta-P, hold down either of the Meta keys and strike P. To type Control-Meta-T hold down both Control and Meta. Unlike ASCII, the Lisp machine character set does not simply label a few of the characters as "control characters"; Control and Meta (and Super and Hyper) are modifiers that can be attached to any character and are represented as separate bits. These modifier bits are not present in characters in strings or files.

Many of the functions refer to "areas". The *area* feature is of interest only to writers of large systems and can be safely disregarded by the casual user. It is described in chapter 16.

1.4 Common Lisp Support

Common Lisp is the name of a standardization project whose goal was to establish a compatible subset for Lisp systems descended from Maclisp.

Originally it was hoped that Zetalisp and the Lisp Machine system could be changed to become a superset of Common Lisp; but this proved impossible because the final Common Lisp design includes several incompatible changes to widely used functions, which, while of no fundamental importance, would make most user programs fail to work. Therefore it was necessary to make Common Lisp a separate mode of operation. The incompatibilities fall into two classes:

* Read syntax: Common Lisp specifies '\' as the single-character escape character rather than the traditional '/'. A few other constructs, such as character objects and complex numbers, are also written incompatibly.
* Specific functions: many Lisp functions of ancient pedigree, including **member**, **assoc**, **subst**, **union**, **terpri**, **close** and **//** are specified to be incompatible with their traditional behavior.

The read syntax incompatibilities have been dealt with by having separate readtables for traditional and Common Lisp syntax. The incompatibilities in functions have been dealt with by means of *reader symbol substitutions*. For each function changed incompatibly, such as **member**, a new, distinct symbol exists in a package called **cli** ("Common Lisp Incompatible"); for example, **cli:member**. The function definition of the symbol **member** is the traditional definition, while that of **cli:member** is the Common Lisp definition. In Common Lisp programs, the reader is directed to replace **member** with **cli:member** wherever it is seen. So traditional and Common Lisp programs both get the **member** functions they expect. Programs written in traditional syntax can refer to the new **cli** functions with explicit **cli:** package prefixes. Programs written in Common Lisp syntax can refer to the traditional symbols with explicit **global:** package prefixes, but this is not expected to be necessary in code.

The symbol replacements are under control of the current readtable, so that the Common Lisp readtable is responsible for causing **cli:close** to replace **close** and so on.

In this manual, the incompatible Common Lisp functions are documented under names starting with **cli:**, the names by which a traditional program could refer to them. Keep in mind that, in Common Lisp programs, the **cli:** would be omitted. A list of symbols which have incompatible Common Lisp substitutes can be found by looking up **cli:** in the function and variable indices.

Traditional read syntax is used nearly everywhere in the manual. This includes the use of '/' as an escape character, the escaping of '/' itself, and not escaping the character '\', which in traditional syntax is not special. It is up to the user to make appropriate modifications to express the same Lisp object in Common Lisp syntax when necessary.

The majority of Common Lisp changes, those that are upward compatible, have been incorporated directly into Zetalisp and are documented in this manual with no special notice.

Common Lisp read syntax and function definitions may be used either in files or interactively.

For listen loops, including Lisp Listener windows, break loops and the debugger, the choice of syntax and function semantics is made by setting the variable **readtable** to the appropriate readtable (see page 536) or most simply by calling the function **common-lisp**.

common-lisp *flag*

If *flag* is **t**, selects Common Lisp syntax and function definitions. If *flag* is **nil**, selects traditional syntax and function definitions.

In either case, this controls the reading of the following expressions that you type in the same process. It works by setting **readtable**.

In a file, Common Lisp is requested by writing the attribute **Readtable: Common-Lisp;** in the `-*-` file's line. This controls both loading or compiling the file and evaluation or compilation in the editor while visiting the file. **Readtable: Traditional;** specifies the use of traditional syntax and function definitions. If neither attribute is present, the file is processed using whatever syntax is selected in the process that loads it. See section 25.5, page 594.

Reading and printing done by programs are controlled by the same things that control reading of programs. They can also be controlled explicitly by binding the variable **readtable**.