Maclisp Extensions

July 1981

Alan Bawden
Glenn S. Burke
Carl W. Hoffman

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Support for this research was provided in part by National Institutes of Health grant number 1 P01 LM 03374-03 from the National Library of Medicine, the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract numbers N00014-75-C-0661 and N00014-77-C-0641, the National Aeronautics and Space Administration under grant NSG 1323, the U. S. Department of Energy under grant ET-78-C-02-4687, and the U. S. Air Force under grant F49620-79-C-020.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LABORATORY FOR COMPUTER SCIENCE

CAMBRIDGE MASSACHUSETTS 02139

Abstract

This document describes a common subset of selected facilities available in Maclisp and its derivatives: PDP-10 and Multics Maclisp, Lisp Machine Lisp (Zetalisp), and NIL. The object of this document is to aid people in writing code which can run compatibly in more than one of these environments.

Acknowledgements

Much of the documentation presented here is drawn from pre-existing sources and modified to be presentable in this context. The documentation on sharpsign is derived from that written by Richard S. Lamson as a Multics online help segment. The descriptions of backquote and defstruct are derived from existing online documentation. The documentation on format shares some portions with the Lisp Machine Manual; text has been exchanged in both directions. The description of defmacro also draws heavily on the existing documentation in the Lisp Machine Manual. The Lisp Machine Manual is authored by Daniel Weinreb and David Moon, and the format documentation therein was contributed to greatly by Guy Steele; they have all thus indirectly contributed a great deal to this paper, as have innumerable others who aided in the preparation of the Lisp Machine Manual.

We would like to thank Joel Moses for providing the motivation to bring Lisp up-to-date on Multics, and Peter Szolovits, under whose auspices this document was produced.

Note

Any comments, suggestions, or criticisms will be welcomed. Please send Arpa network mail to MACLISP-EXTENSIONS@MIT-ML.

Those not on the Arpanet may send U.S. mail to

Glenn S. Burke
Laboratory for Computer Science
545 Technology Square
Cambridge, Mass. 02139

Table of Contents

1. Introduction

1.1 Compatibility

This manual is about compatibility between the PDP-10 and Multics dialects of Maclisp, and the Maclisp derivative Lisps, Lisp Machine Lisp, and NIL.

Believe it or not, it really is possible to write code that runs in all of these Lisp dialects. It is not always a *completely* painless thing to do, but with a little bit of care it is possible to write reasonable code that runs in many places, and that doesn't offend *everyone* who tries to read it.

The biggest stumbling block to writing code that runs in a Lisp dialect other than the one you are most familiar with is the fact each of these Lisps has grown a *different* set of additional features since the original *Maclisp Reference Manual* was written in 1974. How are you supposed to be able to restrain yourself from using all the winning new features that the implementors of your dialect have given you?

Well, unfortunately, you are going to have to avoid some of them. After all, some are probably *impossible* to implement everywhere. On the other hand, some of them are so useful that they have already migrated to all of the places you are planning to move your code. Those are the features that are documented in this manual.

1.2 Conventions

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

The symbol "==>" will be used to indicate macro expansion in examples. Thus, when you see "(foo bar) ==> (aref bar 0)", this means the same thing as "the result of macro-expanding (foo bar) is (or would have been) (aref bar 0)".

Most numbers shown are in octal (base eight). Numbers followed by a decimal point are in decimal (base ten). Despite growing sentiment in favor of decimal as the default base for Lisp reading, it is still the case that most of the Lisps we are concerned with read numbers in octal by default; the sole exception at this time is NIL.

Symbols are consistently written in lower case. This is because on Multics, most symbols have lowercase printnames, and case translation is *not* done by default on input. In the other implementations, where most symbols have uppercase printnames, lowercase characters are translated to uppercase on input, so a symbol *typed* in lowercase will always be read correctly everywhere.

2. Backquote

The backquote facility defines two reader macro characters, backquote ("`", ascii 140) and comma (",", ascii 54). These two macro characters can be used together to abbreviate large compositions of functions like cons, list, list* (page 19) and append. It is typically used to specify templates for building code or other list structure, and often finds application in the construction of Lisp macros.

Backquote has a syntax similar to that of quote ("'", ascii 47). A backquote is followed by a single form. If the form does not contain any use of the comma macro character, then the form will simply be quoted. For example:

```
'(a b c)  =  (quote (a b c))  =  '(a b c)
```

The comma macro character may only be used within a form following a backquote. Comma also has a syntax like that of quote. The comma is followed by a form, and that form is evaluated even though it is inside the backquote. For example:

```
`(,a b c)  =  (cons a (quote (b c)))
           =  (cons a '(b c))

`(a ,b c)  =  (list* (quote a) b (quote (c)))
           =  (list* 'a b '(c))

`(a b ,c)  =  (list (quote a) (quote b) c)
           =  (list 'a 'b c)

`(a . ,rest)  =  (cons (quote a) rest)
              =  (cons 'a rest)
```

In other words, all the components of the backquoted expression are quoted, *except* those preceeded by a comma. Thus, one could write the common macro push using backquote by proceeding from the standard definition

```
(defun push macro (form)
    (list 'setq (caddr form)
          (list 'cons (cadr form) (caddr form))))
```

to

```
(defun push macro (form)
   `(setq ,(caddr form) (cons ,(cadr form) ,(caddr form))))
```

Note how the code to build the macro's output code begins to look more like the output code itself. In fact, with a use of let, we can go all the way to

```
(defun push macro (form)
     (let ((datum (cadr form))
           (list (caddr form)))
       `(setq ,list (cons ,datum ,list))))
```

and produce very legible code. An even better method for defining macros is defmacro (chapter 5, page 10).

Backquote expands into forms that call cons, list, list* or whatever other functions it deems appropriate for the task of constructing a form that looks like the one following the backquote, but with the values of the forms following the commas substituted in.

Since backquote's contract is specified not in terms of the code that it expands into, but rather in terms of what that code produces when evaluated, assumptions should not be made about what the code might look like. The backquote expansions shown in this section are only possible expansions; it is not guaranteed that this is the way they will expand in any particular implementation.

If a comma inside a backquote form is followed by an "at" sign ("@", ascii 100), then the form following the ",@" should return a list. (On Multics, since the default line kill character is @, the user may need to type \@ in order to get lisp to read a @.) Backquote arranges that the elements of that list will be substituted into the resulting list structure. Frequently this involves generating a call to the function append. For example:

```
'(,@a b c)  =  (append a (quote (b c)))
            =  (append a '(b c))

'(a ,@b c)  =  (cons (quote a) (append b (quote (c))))
            =  (cons 'a (append b '(c)))

'(a b ,@c)  =  (list* (quote a) (quote b) c)
            =  (list* 'a 'b c)
```

Similar to following the comma by an atsign is following the comma by a dot (".", ascii 56). The dot is a declaration to backquote telling it that the list returned by the form following the ",." is expendable. This allows backquote to produce code that calls functions like nconc that rplac the list.

Backquote examines the forms following the commas to see if it can simplify the resulting code. For example:

```
'(a b . ,(cons x y))  =  (list* (quote a) (quote b) x y)
                      =  (list* 'a 'b x y)

'(a 3 ,b c ,17)  =  (list* (quote a) 3 b (quote (c 17)))
                 =  (list* 'a 3 b '(c 17))

'(a ,@b ,@nil)  =  (cons (quote a) b)
                =  (cons 'a b)

'(a ,.b ,@(nconc c d))  =  (cons (quote a) (nconc b c d))
                        =  (cons 'a (nconc b c d))
```

These examples should convince the user that he really cannot depend on what the code that backquote expands into will look like. A simple-minded backquote might expand (,@a ,@nil) into (append a 'nil), but this cannot be used as a reliable way to copy a list since a sophisticated backquote can optimize the copying away.

It is sometimes useful to nest one use of backquote within another. This might happen when the user is writing some code that will cons up some more code that will in turn cons up yet more code. The usual example is in writing macro defining macros. When this becomes necessary it is sometimes difficult to determine exactly how to use comma to cause evaluation to happen at the correct times. The following example exhibits all the useful combinations:

```
``(a ,b ,.,c ,',d)
        =  (list 'list* ''a 'b c (list 'quote (list d)))
```

When evaluated once this yields:

```
(list* 'a b <c-at-time-1> '(<d-at-time-1>))
```

Which when evaluated yields:

```
(a <b-at-time-2> <<c-at-time-1>-at-time-2> <d-at-time-1>)
```

Thus "" means never evaluate, "," means evaluate only the second time, ",," means evaluate both times, and ",'," means evaluate only the first time.

3. Sharpsign

The Lisp reader's syntax can be extended with abbreviations introduced by sharp sign ("#", ascii 43). These take the general form of a sharp sign, a second character which identifies the syntax, and following arguments. Certain abbreviations allow a decimal number or certain special "modifier" characters between the sharp sign and the second character. (On Multics, since the default erase character is #, it may be necessary to type \# in order to get lisp to read a #.)

List of # macro abbreviations:

#/char

reads in as the number which is the character code for the character *char*. For example, #/a is equivalent to 141 but clearer in its intent. This is the recommended way to include character constants in your code. Note that the slash causes this construct to be parsed correctly by the Emacs and Zwei editors.

As in strings, upper and lower-case letters are distinguished after #/. Any character works after #/, even those that are normally special to read, such as parentheses. Even non-printing characters may be used, although for them #\ is preferred.

#\name

reads in as the number which is the character code for the non-printing character symbolized by *name*. A large number of character names are recognized; these are documented below. The abbreviations cr for return and sp for space are accepted and generally preferred, since these characters are used so frequently. The rules for reading *name* are the same as those for symbols; the name must be terminated by a delimiter such as a space, a carriage return, or a parenthesis.

#^char

generates Control-*char*. Thus #^*char* always generates the character returned by tyi if the user holds down the control key and types *char*.

#'form

is an abbreviation for (function *form*). *form* is the printed prepresentation of any object. This abbreviation can be remembered by analogy with the ' macro-character, since the function and quote special forms are somewhat analogous.

#,form

evaluates *form* (the printed representation of a Lisp form) at read time, unless the compiler is doing the reading, in which case it is arranged that *form* will be evaluated when the compiled output file is loaded. This is a way, for example, to include in your code complex list-structure constants which cannot be written with quote. Note that the reader does not put quote around the result of the evaluation. You must do this yourself if you want it, typically by using the ' macro-character. An example of a case where you do not want quote around it is when this object is an element of a constant list.

#.form

evaluates *form* (the printed representation of a lisp form) at read time, regardless of who is doing the reading. This abbreviation would be used to supply constant parameters to the compiler. For example, a program might contain #.PI, rather than 3.14159.

#Onumber
reads *number* in octal regardless of the setting of ibase.

#radixRnumber
reads *number* in radix *radix* regardless of the setting of ibase. *radix* must consist of only digits, and it is read in decimal.

For example, #3R102 is another way of writing 11. and #11R32 is another way of writing 35. In Maclisp, supradecimal bases may be used if *number* is preceded by + or -; (status +) is temporarily modified to make this work.

 # +feature
This abbreviation provides a read-time conditionalization facility. It is used as *# +feature form*. If *feature* is a symbol, then this is read as *form* if (status feature *feature*) is true. If (status feature *feature*) is nil, then this is read as whitespace. Alternately, *feature* may be a boolean expression composed of and, or, and not operators and symbols representing items which may appear on the (status features) list. (or lispm amber) represents evaluation of the predicate (or (status feature lispm) (status feature amber)) in the read-time environment.

For example, *# +* lispm *form* makes *form* exist if being read by the Lisp machine. *# +* (or lispm nil) *form* will make *form* exist on either the Lisp machine or in NIL. Note that items may be added to the (status features) list by means of (sstatus feature *feature*), thus allowing the user to selectively interpret or compile pieces of code by parameterizing this list. The most common features checked for using *# +* are: lispm (present on Lisp Machines), Maclisp, NIL, Multics, ITS, TOPS-20 and PDP10.

See also section 11.4.1, page 62 for a more general discussion of conditionalization.

#-feature form
is equivalent to *# +* (not *feature*) *form*.

#M form
is equivalent to *# +* Maclisp *form*.

#Q form
is equivalent to *# +* lispm *form*.

#N form
is equivalent to *# +* NIL *form*.

The following are the recognized special character names, with their synonyms. These names can be used after a "#\" to get the character code for that character.

```
backspace       bs
tab
newline
linefeed        lf
return          cr
formfeed        ff      form
altmode         alt
space           sp
vt
null
help
delete          rubout
```

Certain of these character groupings may overlap in some implementations. For example, on Multics, help is simply the ? character. newline will generally be equivalent to either return or linefeed, as appropriate for the host operating system.

4. Extended Defun

defun *Special Form*

defun is the usual way of defining functions. It still works the way it always has, but several improvements have been added over the years.

A defun form looks like:

```
(defun name lambda-list
  body ...)
```

As in the past, *name* can be a symbol which is to be defined as a function. Alternatively, *name* can be a list of the form (*symbol property*). This arranges to give *symbol* a *property* property of the function, rather than defining some symbol to be that function. In other words, after a defun like

```
(defun (foo bar) (x)
  (cons x x))
```

it would be the case that

```
(funcall (get 'foo 'bar) 34) => (34 . 34)
```

In the simplest case *lambda-list* is a list of variables to bind to the arguments to the function; this is as it has always been. In addition, the keywords &optional, &rest and &aux are allowed to appear there. (Thus these are no longer valid variable names, but nobody seems to have been inconvienced by this.) Their meanings are as follows:

&optional All of the variables following the &optional keyword (and up to the next &-keyword) are optional. Thus a *lambda-list* of the form

```
(a b &optional c d)
```

means that the function may be passed from two to four arguments. a and b are called *required* arguments, c and d are called *optional* arguments (not surprisingly). If an optional argument is not passed in by the caller, then the corresponding variable will be bound to nil. If some other default value is desired, then that value may be specified as follows:

```
(a b &optional (c 'default) (d b))
```

This will bind c to the symbol default if the function is passed only two arguments. If the function is passed less than four, then d will be bound to the second argument. This is because the variables are bound in sequence, so their default values may refer to the values of variables already bound.

It is also possible to find out whether an optional variable was supplied. The bvl

```
(a b &optional (c 'default c-p))
```

will bind the variable c-p to t if the function was passed three arguments (i.e., an argument was supplied for c), nil if it was passed only two.

&rest This keyword must be followed by exactly one variable called the *rest* variable. &rest must also appear *after* any required or optional variables. The rest variable will be bound to a list of the remaining arguments that were passed to the function. For example:

```
(a b &rest c)
```

is the lambda-list to use for a function that accepts two or more arguments. The variable c will be bound to a list of the arguments from the third one on.

```
(a b &optional (c 0) &rest d)
```

would specify that the function takes two are more arguments. If called on exactly two arguments, c will be bound to 0 and d will be bound to nil. If called on three or more arguments, c will be bound to the third argument and d will be bound to a list of the fourth through last argument.

In the Lisp Machine implementation, the rest variable will be bound to a stack allocated list that is only valid during the invocation of that function. This means that the function should not incorporate this list into any permanent data-structure; it should use a copy of the list instead.

In NIL, the rest variable will be bound to a vector which may be stack allocated. &restl instead of &rest selects a list. Unfortunately, &restl is only recognized in PDP-10 Maclisp and NIL.

&aux Following the keyword &aux are some more variables called *auxiliary* variables. &aux must follow all required and optional variables and the rest variable if it is given. Auxiliary variables do not correspond to arguments to the function at all, they are simply local variables that are bound sequentually after the argument variables. For example:

```
(l &optional (a t) &aux (len (length l)) tem)
```

is the lambda-list of a one or two argument function. b will be bound to t if the second argument is not given, then len will be bound to the length of the list that was the first argument, and tem will be bound to nil (presumably for use later on.)

In Maclisp, functions with optional or rest variables will be implemented using the lexpr mechanism. In these implementations it may be necessary to declare these functions as lexprs in order to assure proper compilation.

The syntax

```
(defun name macro (form)
  ...)
```

is still understood as a way to define a macro, but the new macro defining macro **defmacro** is now the prefered way to do so. **defmacro** is documented in chapter 5, page 10.

5. Defmacro

defmacro *Macro*

defmacro is a macro-defining macro which allows one to define macros in a more natural or functional way.

If we want to define the first macro such that (first x) is equivalent to (car x), we could do

```
(defun first macro (x)
   (list 'car (cadr x)))
```

or, using backquote (page 2),

```
(defun first macro (x)
    '(car ,(cadr x)))
```

Just as backquote makes constructing list structure less cumbersome, defmacro allows us to access the "arguments" to a macro in a much cleaner manner. The first macro looks like

```
(defmacro first (l)
   '(car ,l))
```

when defined with defmacro.

In general, the argument list to a macro defined with defmacro is a pattern to be matched against the body of the macro call. The symbols in the pattern will be bound to the corresponding components, and then the body of the macro evaluated, the same as is done for an ordinary macro. That is, for the macro call (first (get 'frob 'elements)), the pattern (l) is matched against ((get 'frob 'elements)), and l gets bound to the form (get 'frob 'elements).

The macro push, which is defined on page 2 as

```
(defun push macro (form)
     (let ((datum (cadr form))
           (list (caddr form)))
       '(setq ,list (cons ,datum ,list))))
```

could be defined with defmacro by

```
(defmacro push (datum list)
   '(setq ,list (cons ,datum ,list)))
```

Macros, and thus defmacro, are useful for defining forms which provide syntax for some kind of control structure. For example, someone might want a limited iteration construct which increments a variable by one until it exceeds a limit (like the FOR statement of the BASIC language). One might want it to look like

```
(for a 1 100 (print a) (print (* a a)))
```

To get this, one could write a macro to translate it into

```
(do a 1 (1+ a) (> a 100) (print a) (print (* a a)))
```

A macro to do this could be defined with

```
(defun for macro (x)
   '(do ,(cadr x) ,(caddr x) (1+ ,(cadr x))
        (> ,(cadr x) ,(cadddr x))
      ,@(cddddr x)))
```

Alternatively, for could be defined with defmacro:

```
(defmacro for (var lower upper . body)
   '(do ,var ,lower (1+ ,var) (> ,var ,upper)
      ,@body))
```

If a pattern is not sufficient, or if a more function-like interface is desired, the argument list to defmacro may contain certain &-keywords. These are analogous to the &-keywords accepted by defun (see page 8). In this case, the argument list should *not* have a dotted end (like the for example), although the components may themselves be patterns.

&optional denotes the start of optional "arguments" to the macro. Each following parameter is then of the form *variable*, (*variable*), (*variable default*), or (*variable default present-p*). *default* is a form to be evaluated to provide a value of no corresponding "argument" is present in the call. *present-p* is a variable; it will be bound to nil if no argument is present, t otherwise. For example,

```
(defmacro print-in-radix (x &optional (radix 10.) (*nopoint? t))
   '(let ((base ,radix) (*nopoint ,*nopoint?))
        (print ,x)))
```

If *variable* is a pattern, then the first form is disallowed byecause it is syntactically ambiguous. The pattern must be enclosed in a singleton list. Note: in some implementations, if *variable* is a pattern, *default* may be evaluated more than once.

&rest says that the following item should be matched against the rest of the call. That is, the argument list (&rest items) is equivalent to the argument list items, and the argument list for for, (var lower upper . body), could have been written as (var lower upper &rest body). &rest may be easier to read than a dotted list, and it allows one to use &aux.

&aux has nothing to do with pattern matching. It should come at the end of the pattern (which thus cannot be a dotted list), and may be followed by one or more variable binding specifications, of the form *variable* or (*variable value*). The variable will be bound to the specified value, or nil.

&body is identical to &rest, and in certain implementations may leave some information around for other programs to use to decide on how that form should be indented. The for macro should be defined with &body in preference to &rest.

The &optional variable bindings are performed sequentually. Thus something like

```
(defmacro foo (a &optional (b a)) ...)
```

will define a macro that when called with only one argument will bind both a and b to that argument. When called with two arguments a will be bound to the first argument, and b will be

bound to the second.

The macro dolist (page 23) is defined such that

`(dolist (`*var list*`)` *form-1 form-2 ...*`)`

steps *var* over the elements of *list*, evaluating all of the *form-i* each time (sort of like mapc). It could be defined with defmacro by

```
(defmacro dolist ((var list) &body forms
                  &aux (list-var (gensym)))
  '(do ((,list-var ,list (cdr ,list-var))
        (,var))
       ((null ,list-var))
     (setq ,var (car ,list-var))
     ,@forms))
```

6. Other Definition Facilities

defvar *variable* [*init*] [*documentation*] *Special Form*

defvar is the recommended way to declare the use of a global variable in a program. The form

```
(defvar variable init)
```

placed at top level in a file is roughly equivalent to

```
(declare (special variable))
(or (boundp 'variable)
    (setq variable init))
```

If the *init* form is not given, then defvar does not try to initialize the value of the variable, it only declares it to be special.

documentation is ignored in most implementations, although it is a good idea to supply it for the benifit of those implementations that make use of it. It should be a "string" (see page 63).

defconst *variable* [*init*] [*documentation*] *Special Form*

defconst is similar to defvar expect that if *init* is given, then *variable* is *always* set to have that value, regardless of whether it is already bound. The idea is that defvar declares a global variable, whose value is initialized to something but will then be changed during the running of the program. On the other hand, defconst declares a constant, whose value will never be changed by the program, only by changes *to* the program. defconst always sets *variable* to the specified value so that if you change your mind about what the constant value should be, and then you evaluate the defconst form again, *variable* will get set to the new value.

eval-when *times-list forms...* *Special Form*

eval-when is used to specify precisely what is to happen to the containing forms. An eval-when form must appear at top level in a file. *times-list* can contain any combination of the symbols eval, compile and load.

If eval is in *times-list*, then when the interpreter evaluates the eval-when form each of the forms will be evaluated. If eval is not present, then the forms will be ignored in the interpreter. The return value is not guaranteed to be anything in particular.

If compile is in *times-list*, then when the compiler comes across the eval-when form at compile-time, it will evaluate each of the forms right then and there.

If load is in *times-list*, then when the compiler comes across the eval-when form in the file, it will continue process the forms as if they appeared at top level in the file. Thus the result of compiling the forms will be placed into the compiler output file so that they may be loaded later.

Examples:

```
(eval-when (eval compile)
  (setsyntax /" 'macro 'hack-strings)
  (defun hack-strings ()
    ...))
```

This will fool with the syntax of doublequote at run-time and compile-time (presumably to allow the rest of the file to be read in properly), but when the file is compiled and loaded the syntax of doublequote will be unchanged, and the function hack-strings will not be defined.

```
(eval-when (eval)
  (defun foo (frob)
    (and (atom frob) (barf))
    (car frob)))

(eval-when (compile)
  (defun foo macro (x)
    (list 'car (cadr x))))
```

This will define foo as a paranoid error checking function when the program is being run interpreted, but will arrange to define foo as a macro at compile-time so that it will compile just like car. When the compiled file is loaded foo will not be defined at all.

```
(eval-when (eval compile load)
  (defprop frobulate frobulate-macro macro)
  (defun frobulate-macro (x)
    ...))
```

This is a way to define a macro by hand in Maclisp to be present whenever the file is being run or compiled.

7. Setf

setf *Macro*

setf provides a general mechanism for modifying the components of arbitrary Lisp objects. A setf form looks like:

```
(setf reference form)
```

The setf form expands into code to evaluate *form* and then modify some Lisp object such that the form *reference* would evaluate to the same thing. For example:

```
(setf (car x) 47)           ==> (rplaca x 47)
(setf (cadr x) nil)         ==> (rplaca (cdr x) nil)
(setf (get a 'zip) 'foo)    ==> (putprop a 'foo 'zip)
(setf (arraycall t a 1) t) ==> (store (arraycall t a 1) t)
(setf (symeval foo) bar)    ==> (set foo bar)
(setf foo bar)              ==> (setq foo bar)
```

The order in which *form* and any forms found in *reference* are evaluated is not guaranteed in any but the PDP-10 Maclisp and NIL implementations of setf. Neither is the value returned by the code setf expands into guaranteed in any way.

setf also knowns how to perform macro expansions of any *reference* it doesn't recognize. So if first is a macro defined to expand as

```
(first foo)     ==> (car foo)
```

then

```
(setf (first foo) t)    ==> (rplaca foo t)
```

setf's ability to expand macro forms makes it indispensable when using the defstruct macro (page 26).

Several other common macros are defined to expand into code that includes a setf form. All these other macros share the property with setf that in some implementations they are liable to evaluate their various sub-forms in an order other than the one they were written in. In some cases you even run the risk of having some sub-form evaluated more that once.

push *Macro*

push is defined to expand roughly as follows:

```
(push frob reference)

==> (setf reference (cons frob reference))
```

The qualifications about order of evaluation given for setf apply to push also; additionally, only the PDP-10 and NIL implementations guarantee that forms in *reference* will not be evaluated multiple times.

pop *Macro*

pop is defined to expand roughly as follows:

```
(pop reference)

==> (prog1 (car reference)
           (setf reference (cdr reference)))
```

(prog1 is explained on page 25.)

The qualifications given for push about order of evaluation and multiple evaluation apply to pop also.

8. New Functions and Special Forms

This chapter documents a number of new functions and special forms that have been added to the Maclisp language.

Although many of the functions documented here are shown shown as being functions, there is no guarantee that any particular Lisp actually implements them that way, rather than as macros.

8.1 Bit Hacking

All of the functions in this section operate on integers of any size in Lisp Machine Lisp, but only on fixnums elsewhere. Remember that all the integers shown here are in octal.

8.1.1 Boolean Operations

The following functions could be (and often are) implemented in terms of the boole function. Their use tends to produce less obscure code.

logand &rest *args*
Returns the bit-wise logical *and* of its arguments. At least two arguments are required.
Examples:

```
(logand 3456 707)  => 406
(logand 3456 -100)  => 3400
```

logior &rest *args*
Returns the bit-wise logical *inclusive or* or its arguments. At least two arguments are required.
Example:

```
(logior 4002 67)  => 4067
```

logxor &rest *args*
Returns the bit-wise logical *exclusive or* of its arguments. At least two arguments are required.
Example:

```
(logxor 2531 7777)  => 5246
```

lognot *number*
Returns the logical complement of *number*. This is the same as logxor'ing *number* with -1.
Example:

```
(lognot 3456)  => -3457
```

8.1.2 Byte Manipulation

Several functions are provided for dealing with an arbitrary-width field of contiguous bits appearing anywhere in an integer (in Maclisp, this is restricted to a fixnum). Such a contiugous set of bits is called a *byte*. Note that the term *byte* is not being used to mean eight bits, but rather any number of bits within an integer. These functions use numbers called *byte specifiers* to designate a specific byte position within any word. Byte specifiers are fixnums whose two lowest octal digits represent the *size* of the byte, and whose higher octal digits represent the *position* of the byte within a number, counting from the right in bits. A position of zero means that the byte is at the right end of the number. For example, the byte-specifier 0010 (i.e., 10 octal) refers to the lowest eight bits of a word, and the byte-specifier 1010 refers to the next eight bits. These byte-specifiers will be stylized below as *ppss*. The maximum reasonable values of *pp* and *ss* are dictated by the Lisp implementation, except of course *ss* may not "overflow" into the *pp* field, so may not exceed 77 (octal).

ldb *ppss num*
Returns the byte of *num* specified with the byte-specifier *ppss*, as described above. Example:

```
(ldb 0306 4567)  => 56
```

dpb *byte ppss num*
Returns a new number made by replacing the *ppss* byte of *num* with *byte*.

8.1.3 Testing

bit-test *x y*
Returns t if any of the bits in *x* and *y* intersect; that is, if their logand is not zero. bit-test could be (and sometimes is) defined as a macro such that

```
(bit-test x y)  ==>  (not (zerop (logand x y)))
```

8.2 Predicates

fixnump *x*
Returns t if *x* is a fixnum. This corresponds to a typep of fixnum.
Examples:

```
(fixnump 1)  =>  t
(fixnump (expt 259. 259))  =>  nil
```

flonump *x*
Returns t if *x* is a flonum. This corresponds to a typep of flonum.
Examples:

```
(flonump 3.14)  =>  t
(flonump 17)  =>  nil
```

Note that this is the same as floatp in most Lisps, which have only one type of floating-point representation. In Lisp Machine Lisp however, there are some kinds of floating

point numbers that are *not* of type flonum. flonump will return nil for these objects. It is probably the case that code that is trying to be compatible should use floatp in preference to either flonump or (eq (typep *x*) 'flonum).

arrayp *x*
Returns t if *x* is an array. Note that some Lisps implement certain kinds of objects as arrays; for example, PDP-10 Maclisp *file objects* are arrays, and Lisp Machine Lisp utilizes arrays for most structures defined with defstruct (page 26).

evenp *integer*
Returns t if *integer* is even, nil otherwise. This complements the oddp function which Lisp provides.

<= &rest *args*
<= requires at least two arguments. If any argument is greater than the next argument, it returns nil, otherwise it returns t. In Maclisp, *args* should consist of either all fixnums or all flonums.

>= &rest *args*
Similar to <=.

fboundp *symbol*
fboundp returns nil if the symbol *symbol* in *not* defined as a function or special form. It returns something non-nil if *symbol* is defined. The exact nature of the non-nil object varies from implementation to implementation.

It is not defined what fboundp returns if *symbol* has an autoload property and is otherwise undefined.

8.3 Lists

list* &rest *args*
list* creates what some people call a "dotted list".

```
(list* 'foo 'bar 'baz) => (foo bar . baz)
(list* 'foo 'bar)      => (foo . bar)
(list* 'foo)           => foo
```

list* makes certain unwieldy compositions of the cons function somewhat easier to type:

```
(list* 1 2 3 4)
```

is the same as

```
(cons 1 (cons 2 (cons 3 4)))
```

make-list *length*
make-list creates a list of nils of length *length*. Example:

```
(make-list 3) => (nil nil nil)
```

nth *n list*

(nth *n list*) returns the *n*'th element of *list*, where the zeroth element is the car of the list. If *n* is larger than the length of the list, nth returns nil. Examples:

```
(nth 2 '(zero one two three)) => two
(nth 0 '(a b c)) => a
```

nthcdr *n list*

(nthcdr *n list*) cdrs *list n* times, and returns the result. If *n* is larger than the length of the list then nil is returned. Examples:

```
(nthcdr 3 '(q w e r t y)) => (r t y)
(nthcdr 0 '(e t a o i n r)) => (e t a o i n r)
```

Note that

```
(nth n l)
```

is the same as

```
(car (nthcdr n l))
```

8.4 Variables

let *Special Form*

```
(let ((var-1 val-1) (var-2 val-2) ...)
  form-1
  form-2
  ...)
```

binds *var-1* to the value of *val-1*, *var-2* to the value of *val-2* etc., and evaluates each of the *form-i* in that binding environment. That is, it is equivalent to

```
((lambda (var-1 var-2 ...)
     form-1 form-2 ...)
 val-1 val-2 ...)
```

but displays the values in close proximity to the variables.

Note that similar to do, a declaration is allowed as the first form in a let body.

let* *Special Form*

let* has a syntax identical to that of let, but binds the variables in sequence rather than in parallel. Thus,

```
(let* ((a (foo)) (b (bar a)))
   (computate a b))
```

is like

```
((lambda (a)
    ((lambda (b)
        (computate a b))
     (bar a)))
 (foo))
```

psetq *Special Form*

psetq is similar to setq. In the multi-variable case however, the variables are set "in parallel" rather than sequentually; first all the forms are evaluated, and then the symbols are set to the resulting values. For example:

```
(setq a 1)
(setq b 2)
(psetq a b b a)
a => 2
b => 1
```

8.5 Flow of Control

8.5.1 Conditionals

if *predicate-form then-form* [*else-form*] *Special Form*

if is a convenient abbreviation for a simple cond which does a binary branch. *predicate-form* is evaluated, and if the result is non-nil, then *then-form* is evaluated and that result returned, otherwise *else-form* is evaluated and that result returned. If no else-form is specified and *predicate-form* evaluates to nil, then nil is returned. if can (and usually is) defined as a macro such that

```
(if pred then else)
    ==>  (cond (pred then) (t else))
```

and

```
(if pred then)
    ==>  (cond (pred then) (t nil))
```

or

```
    ==>  (and pred then)
```

If there are more than three subforms, if assumes that more than one *otherwise* form was intended; they will be treated as an implicit progn. For example,

```
(if p c e1 e2 e3)
  ==> (cond (p c) (t e1 e2 e3))
```

There is disagreement as to whether this constitutes good programming style, so it is possible that this last variant may be disallowed.

8.5.2 Selection

selectq *key-form clauses...* *Special Form*

selectq is a conditional which chooses one of its clauses to execute by comparing the value of a form against various constants. Its form is as follows:

```
(selectq key-form
   (test consequent-forms...)
   (test consequent-forms...)
   ...)
```

The first thing selectq does is to evaluate *key-form*; call the resulting value *key*. Then selectq considers each of the clauses in turn. If *key* matches the clauses *test*, the consequents of this clause are evaluated, and selectq returns the value of the last consequent. If there are no matches, selectq returns nil.

A *test* may be any of

a symbol or integer — The symbol or integer is compared with *key*. Symbols are compared using eq; integers are compared on the same basis that equal uses—equal types and equal values. Note that t and otherwise are exceptions here.

a list — The list should contain only symbols and integers, which are compared as above.

t or otherwise — The symbols t and otherwise are special keywords which match anything. Either of these may thus be used to signify a "default" clause, which to be useful, should be the last clause of the selectq.

Examples:

```
(defun count-them (n)
  (selectq n
    (0 'none)
    (1 'one)
    (2 'two)
    ((3 4) 'a-few)
    (t 'many)))
(count-them 2)  =>  two
(count-them 3)  =>  a-few
(count-them 7)  =>  many

(selectq 'one
   (1 integer-one)
   (one 'symbol-one)
   (t 'something-else))
  => symbol-one
```

If the keys being tested against and the value of *key-form* are all of the same type, caseq should be used, as it may produce more efficient code depending on the implementation. This is true in PDP-10 Maclisp, which has no primitive predicate that implements the type of comparison that selectq uses. In Lisp Machine Lisp and Multics Maclisp there should be no difference unless bignums are used. Presently, bignums do not work

anyway, but this is expected to be fixed.

caseq *key-form clauses...* *Special Form*

caseq is the same as selectq *except* that it requires all of the keys being compared to be of the same type. It is also an error for the value of *key-form* to be of a different type than the keys in the clauses.

Currently, in all but the PDP-10 implementation, caseq is implemented in terms of selectq so does not provide this consistency checking, any qualifications given for selectq apply to caseq.

In PDP-10 Maclisp, caseq does not accept the otherwise keyword; it is necessary for t to be used. It also does not accept bignums.

8.5.3 Iteration

dolist *Special Form*

dolist is like a cross between mapc and do.

```
(dolist (var list) body...)
```

evaluates the forms of *body* for each element of *list*, with *var* bound to the successive elements. *body* is treated as a prog or do body, so it may contain prog tags, and calls to return, which will return from the dolist.

dotimes *Special Form*

dotimes performs integer stepping, and is otherwise similar to dolist.

```
(dotimes (var count) body...)
```

evaluates *body count* times; *var* takes on values starting with zero, and stops before reaching *count*. For example,

```
(dotimes (i (// m n)) (frob i))
```

is equivalent to

```
(do ((i 0 (1+ i))
     (count (// m n)))
    ((not (< i count)))
  (frob i))
```

except that the name count is not used.

dotimes is similar to dolist in that the body is treated as if it were a prog or do body.

loop *Macro*

dolist and dotimes are convenient for simple cases, where the extra syntax necessitated by mapc or do is an annoyance. For complicated cases, the loop macro may be desirable. It provides for the stepping of multiple variables, either in sequence or in parallel, and methods for performing various sorts of accumulations, such as collecting a list, summing, and counting; more than one such accumulation to be performed, and they need not be accumulated "in sync" with the iteration. For example,

```
(loop for x in l as y = (f x) collect (cons x y))
```

produces a result like

```
(do ((*list* l (cdr *list*)) (x) (y) (*result*))
    ((null *list*) (nreverse *result*))
  (setq x (car *list*))
  (setq y (f x))
  (setq *result* (cons (cons x y) *result*)))
```

does. loop is extremely complicated so is not documented here; full documentation may be found in MIT Laboratory for Computer Science Technical Memo 169 (January 1981).

8.5.4 Non-Local Exits

***catch** *tag form* *Special Form*

The *catch special form is used with *throw to perform non-local exits. *tag* is evaluated, and then *form* is evaluated. If during the evaluation of *form* a (*throw *tag value*) is done, then the *catch returns *value*.

***throw** *tag value*

Evaluation of (*throw *tag value*) causes a pending *catch of *tag* to return *value*.

*catch and *throw are slightly more general versions of the standard Maclisp catch and throw special forms. They are more general in that the tags given to them are evaluated, and thus need not be written into the code, but can be passed in. Additionally, the difference in argument ordering can make for more readable code, viz

```
(*catch 'exit
  moby-big-hairy-compuation-
  that-is-continued-over-
  many-lines)
```

Lisp Machine Lisp, PDP-10 Maclisp, and NIL support *catch and *throw as the basic catching and throwing primitives; catch and throw are implemented as macros in terms of them. Multics Maclisp implements *catch and *throw as macros in terms of the existing catch and throw special forms; thus it is impossible for *catch and *throw on Multics to accept anything but a quoted atom for the *tag*.

It is advisable for *catch and *throw to be used in preference to catch and throw; at some future time it is anticipated that catch and throw will be changed to be equivalent to *catch and *throw. The names *catch and *throw are expected to remain valid indefinitely.

unwind-protect *form cleanup-forms...* *Special Form*

unwind-protect evaluates *form* and returns that result as its value. When control returns from the unwind-protect for any reason, whether it be a normal return, or a non-local exit causes by a *throw or an error, the *cleanup-forms* will be evaluated. unwind-protect can thus be used for "binding" something which is not really bindable as a variable, or for performing some necessary cleanup action, such as closing a file.
Example:

```
(unwind-protect
   (progn (turn-on-water-faucet)
          (compute-under-running-water))
   (turn-off-water-faucet))
```

8.6 Miscellaneous

prog1 *first forms...* *Special Form*

prog1 is similar to prog2, only without the first argument. All of the argument to **prog1** are evaluated just as they would be for progn, however, the value returned by **prog1** will be the value of the *first* form rather than the last. For example:

```
(rplaca x (prog1 (cdr x) (rplacd x (car x))))
```

can be used to exchange the car and the cdr of a cons.

lexpr-funcall *function* &rest *args*

lexpr-funcall is a cross beween funcall and apply. (lexpr-funcall *function arg-1 arg-2 ... arg-n list*) calls the function *function* on *arg-1* through *arg-n* followed by the elements of *list*, for example

```
(lexpr-funcall 'list 'a 'b '(c d)) => (a b c d)
(lexpr-funcall 'plus 3 4 '(2 1 0)) => 12
```

Note that two argument lexpr-funcall is the same as **apply**, and that lexpr-funcall with a list argument of nil is essentially **funcall**.

without-interrupts *forms...* *Special Form*

This provides a convenient way of executing some code uninterruptibly. *forms* are evaluated as with progn and the value of the last form is returned. It is guaranteed that the evaluation will be performed as an atomic operation.

ferror *condition-name format-string* &rest *format-args*

ferror provides a mechanism for signalling errors using **format** (page 47) to generate the error message. *condition-name* is used to specify the type of condition which is to be signaled; no mechanism for this exists in Maclisp. However, *condition-name* may be nil, in which case an uncorrectable error occurs—nil is therefore the only value of *condition-name* guaranteed to work everywhere.

Example:

```
(ferror nil "%%% Compiler error - call ~S %%%"
        (get 'compiler 'maintainer))
```

9. Defstruct

9.1 Introduction

The features of **defstruct** differ slightly from one Lisp implementation to another. However, defstruct makes it fairly easy to write compatible code if the user doesn't try to exercise any of the more esoteric features of his particular Lisp implementation. The differences will be pointed out as they occur.

One difference that we must deal with immediately is the question of packages. **defstruct** makes use of a large number of keywords, and on the Lisp Machine those keywords are all interned on the keyword package. However, for the purposes of compatibility, the Lisp Machine defstruct will allow the keywords to appear in any package. The Lisp Machine programmer is discouraged from writing keywords without colons, unless the code is to be transported to another Lisp implementation. Classes of symbols that defstruct treats as keywords will be noted as they occur.

Other package related issues will be dealt with later.

9.2 A Simple Example

defstruct *Macro*

defstruct is a macro defining macro. The best way to explain how it works is to show a sample call to defstruct, and then to show what macros are defined and what each of them does.

Sample call to defstruct:

```
(defstruct (elephant (type list))
  color
  (size 17.)
  (name (gensym)))
```

This form expands into a whole rat's nest of stuff, but the effect is to define five macros: color, size, name, make-elephant and alter-elephant. Note that there were no symbols make-elephant or alter-elephant in the original form, they were created by defstruct. The definitions of color, size and name are easy, they expand as follows:

```
(color x) ==> (car x)
(size x)  ==> (cadr x)
(name x)  ==> (caddr x)
```

You can see that defstruct has decided to implement an elephant as a list of three things: its color, its size and its name. The expansion of make-elephant is somewhat harder to explain, let's look at a few cases:

```
(make-elephant)                      ==> (list nil 17. (gensym))
(make-elephant color 'pink)          ==> (list 'pink 17. (gensym))
(make-elephant name 'fred size 100) ==> (list nil 100 'fred)
```

As you can see, make-elephant takes a "setq-style" list of part names and forms, and expands into a call to list that constructs such an elephant. Note that the unspecified parts get defaulted to pieces of code specified in the original call to defstruct. Note also that the order of the setq-style arguments is ignored in constructing the call to list. (In the example, 100 is evaluated before 'fred even though 'fred came first in the make-elephant form.) Care should thus be taken in using code with side effects within the scope of a make-elephant. Finally, take note of the fact that the (gensym) is evaluated *every time* a new elephant is created (unless you override it).

The explanation of what alter-elephant does is delayed until section 9.4.3, page 31.

So now you know how to construct a new elephant and how to examine the parts of an elephant, but how do you change the parts of an already existing elephant? The answer is to use the setf macro (chapter 7, page 15).

```
(setf (name x) 'bill)   ==> (rplaca (cddr x) 'bill)
```

which is what you want.

And that is just about all there is to defstruct; you now know enough to use it in your code, but if you want to know about all its interesting features, then read on.

9.3 Syntax of defstruct

The general form of a defstruct form is:

```
(defstruct (name option-1 option-2 ... option-n)
  slot-description-1
  slot-description-2
  ...
  slot-description-m)
```

name must be a symbol, it is used in constructing names (such as "make-elephant") and it is given a defstruct-description property of a structure that describes the structure completely.

Each *option-i* is either the atomic name of an option, or a list of the form (*option-name arg . rest*). Some options have defaults for *arg*; some will complain if they are present without an argument; some options complain if they are present *with* an argument. The interpretation of *rest* is up to the option in question, but usually it is expected to be nil.

Each *slot-description-j* is either the atomic name of a slot in the structure, or a list of the form (*slot-name init-code*), or a list of byte field specifications. *init-code* is used by constructor macros (such as make-elephant) to initialize slots not specified in the call to the constructor. If the *init-code* is not specified, then the slot is initialized to whatever is most convenient. (In the elephant example, since the structure was a list, nil was used. If the structure had been a fixnum array, such slots would be filled with zeros.)

A byte field specification looks like: (*field-name ppss*) or (*field-name ppss init-code*). Note that since a byte field specification is always a list, a list of byte field specifications can never be confused with the other cases of a slot description. The byte field feature of defstruct is explained in detail in section 9.5, page 38.

9.4 Options to defstruct

The following sections document each of the options defstruct understands in detail.

On the Lisp Machine, all these defstruct options are interned on the keyword package.

9.4.1 type

The type option specifies what kind of lisp object defstruct is going to use to implement your structure, and how that implementation is going to be carried out. The type option is illegal without an argument. If the type option is not specified, then defstruct will choose an appropriate default (hunks on PDP-10s, arrays on Lisp Machines and lists on Multics). It is possible for the user to teach defstruct new ways to implement structures, the interested reader is referred to section 9.8, page 42, for more information. Many useful types have already been defined for the user. A table of these "built in" types follows: (On the Lisp Machine all defstruct types are interned on the keyword package.)

list *All implementations*

Uses a list. This is the default on Multics.

named-list *All implementations*

Like list, except the car of each instance of this structure will be the name symbol of the structure. This is the only "named" structure type defined on Multics. (See the named option documented in section 9.4.7, page 35.)

tree *All implementations*

Creates a binary tree out of conses with the slots as leaves. The theory is to reduce car-cdring to a minimum. The include option (section 9.4.6, page 33) does not work with structures of this type.

list* *All implementations*

Similar to list, but the last slot in the structure will be placed in the cdr of the final cons of the list. Some people call objects of this type "dotted lists". The include option (section 9.4.6, page 33) does not work with structures of this type.

array *All implementations*

Uses an array object (*not* a symbol with an array property). This is the default on Lisp Machines. Lisp Machine users may want to see the make-array option documented in section 9.4.8, page 35.

fixnum-array *All implementations*

Like array, except it uses a fixnum array and thus your structure can only contain fixnums. On Lisp Machines defstruct uses an art-32b type array for this type.

flonum-array *All implementations*

Analogous to fixnum-array. On Lisp Machines defstruct uses an art-float type array for this type.

un-gc-array *PDP-10 only*

Uses a nil type array instead of a t type. Note that this type does not exist on Lisp Machines or Multics, because un-garbage-collected arrays do not work in those implementations.

hunk *PDP-10 only*

Uses a hunk. This is the default on PDP-10s.

named-hunk *PDP-10 only*

Like hunk, except the car of each instance of this structure will be the name symbol of the structure. This *can* be used with the (status usrhunk) feature of PDP-10 Maclisp to give the user Lisp Machine-like named structures. (See the named option documented in section 9.4.7, page 35.)

sfa *PDP-10 only*

Uses an SFA. The constructor macros for this type accept the keywords sfa-function and sfa-name. Their arguments (evaluated, of course) are used, respectively, as the function and the printed representation of the SFA. See also the sfa-function (section 9.4.9, page 35) and sfa-name (section 9.4.10, page 36) options.

named-array *Lisp Machine only*

Uses an array with the named structure bit set and stores the name symbol of the structure in the first element. (See the make-array option documented in section 9.4.8, page 35.)

array-leader *Lisp Machine only*

Uses an array with a leader. (See the make-array option documented in section 9.4.8, page 35.)

named-array-leader *Lisp Machine only*

Uses an array with a leader, sets the named structure bit, and stores the name symbol in element 1 of the leader. (See the make-array option documented in section 9.4.8, page 35.)

fixnum *All implementations*

This type allows one to use the byte field feature of defstruct to deal symbolically with fixnums that aren't actually stored in any structure at all. Essentially, a structure of type fixnum has exactly one slot. This allows the operation of retrieving the contents of that slot to be optimized away into the identity operation. See section 9.5, page 38 for more information about byte fields.

external *Multics only*

Uses an array of type external (only Multics Lisp has these). Constructor macros for structures of this kind take the external-ptr keyword to tell them where the array is to be allocated. (See section 9.4.2, page 30, for an explanation of constructor macro keywords.) See also the external-ptr option described in section 9.4.11, page 36.

9.4.2 constructor

The constructor option specifies the name to be given to the constructor macro. Without an argument, or if the option is not present, the name defaults to the concatenation of "make-" with the name of the structure. If the option is given with an argument of nil, then no constructor is defined. Otherwise the argument is the name of the constructor to define. Normally the syntax of the constructor defstruct defines is:

```
(constructor-name
    keyword-1 code-1
    keyword-2 code-2
    ...
    keyword-n code-n)
```

Each *keyword-i* must be the name of a slot in the structure (not necessarily the name of an accessor macro; see the conc-name option, section 9.4.5, page 33), or one of the special keywords allowed for the particular type of structure being constructed. For each keyword that is the name of a slot, the constructor expands into code to make an instance of the structure using *code-i* to initialize slot *keyword-i*. Unspecified slots default to the forms given in the original defstruct form, or, if none was given there, to some convenient value such as nil or 0.

For keywords that are not names of slots, the use of the corresponding code varies. Usually it controls some aspect of the instance being constructed that is not otherwise constrained. See, for example, the make-array option (section 9.4.8, page 35), the sfa-function option (section 9.4.9, page 35, or the external-ptr option (section 9.4.11, page 36).

On the Lisp Machine all such constructor macro keywords (those that are *not* the names of slots) are interned on the keyword package.

If the constructor option is given as (constructor *name arglist*), then instead of making a keyword driven constructor, defstruct defines a "function style" constructor. The *arglist* is used to describe what the arguments to the constructor will be. In the simplest case something like (constructor make-foo (a b c)) defines make-foo to be a three argument constructor macro whose arguments are used to initialize the slots named a, b and c.

In addition, the keywords &optional, &rest and &aux are recognized in the argument list. They work in the way you might expect, but there are a few fine points worthy of explanation:

```
(constructor make-foo
        (a &optional b (c 'sea) &rest d &aux e (f 'eff)))
```

This defines make-foo to be a constructor of one or more arguments. The first argument is used to initialize the a slot. The second argument is used to initialize the b slot. If there isn't any second argument, then the default value given in the body of the defstruct (if given) is used

instead. The third argument is used to initialize the c slot. If there isn't any third argument, then the symbol sea is used instead. The arguments from the fourth one on are collected into a list and used to initialize the d slot. If there are three or less arguments, then nil is placed in the d slot. The e slot is *not initialized*. It's value will be something convenient like nil or 0. And finally the f slot is initialized to contain the symbol eff.

The b and e cases were carefully chosen to allow the user to specify all possible behaviors. Note that the &aux "variables" can be used to completely override the default initializations given in the body.

Since there is so much freedom in defining constructors this way, it would be cruel to only allow the constructor option to be given once. So, by special dispensation, you are allowed to give the constructor option more than once, so that you can define several different constructors, each with a different syntax.

Note that even these "function style" constructors do not guarantee that their arguments will be evaluated in the order that you wrote them.

9.4.3 alterant

The alterant option defines a macro that can be used to change the value of several slots in a structure together. Without an argument, or if the option is not present, the name of the alterant macro defaults to the concatenation of "alter-" with the name of the structure. If the option is given with an argument of nil, then no alterant is defined. Otherwise the argument is the name of the alterant to define. The syntax of the alterant macro defstruct defines is:

```
(alterant-name code
   slot-name-1 code-1
   slot-name-2 code-2
   ...
   slot-name-n code-n)
```

code should evaluate to an instance of the structure, each *code-i* is evaluated and the result is made to be the value of slot *slot-name-i* of that structure. The slots are all altered in parallel after all code has been evaluated. (Thus you can use an alterant macro to exchange the contents to two slots.)
Example:

```
(defstruct (lisp-hacker (type list)
                        conc-name
                        default-pointer
                        alterant)
  (favorite-macro-package nil)
  (unhappy? t)
  (number-of-friends 0))

(setq lisp-hacker (make-lisp-hacker))
```

Now we can perform a transformation:

```
(alter-lisp-hacker lisp-hacker
        favorite-macro-package 'defstruct
        number-of-friends 23.
        unhappy? nil)

==> ((lambda (G0009)
       ((lambda (G0011 G0010)
          (setf (car G0009) 'defstruct)
          (setf (caddr G0009) G0011)
          (setf (cadr G0009) G0010))
        23.
        nil))
     lisp-hacker)
```

Although it appears from this example that your forms will be evaluated in the order in which you wrote them, this is not guaranteed.

Alterant macros are particularly good at simultaneously modifying several byte fields that are allocated from the same word. They produce better code than you can by simply writing consecutive setfs. They also produce better code when modifying several slots of a structure that uses the but-first option (section 9.4.15, page 37).

9.4.4 default-pointer

Normally the accessors are defined to be macros of exactly one argument. (They check!) But if the default-pointer option is present then they will accept zero or one argument. When used with one argument, they behave as before, but given no arguments, they expand as if they had been called on the argument to the default-pointer option. An example is probably called for:

```
(defstruct (room (type tree)
                 (default-pointer **current-room**))
  (room-name 'y2)
  (room-contents-list nil))
```

Now the accessors expand as follows:

```
(room-name x)          ==> (car x)
(room-name)            ==> (car **current-room**)
```

If no argument is given to the default-pointer option, then the name of the structure is used as the "default pointer". default-pointer is most often used in this fashion.

9.4.5 conc-name

Frequently all the accessor macros of a structure will want to have names that begin the same way; usually with the name of the structure followed by a dash. The conc-name option allows the user to specify this prefix. Its argument should be a symbol whose print name will be concatenated onto the front of the slot names when forming the accessor macro names. If the argument is not given, then the name of the structure followed by a dash is used. If the conc-name option is not present, then no prefix is used. An example illustrates a common use of the conc-name option along with the default-pointer option:

```
(defstruct (location default-pointer
                    conc-name)
  (x 0)
  (y 0)
  (z 0))
```

Now if you say

```
(setq location (make-location x 1 y 34 z 5))
```

it will be the case that

```
(location-y)
```

will return 34. Note well that the name of the slot ("y") and the name of the accessor macro for that slot ("location-y") are different.

9.4.6 include

The include option inserts the definition of its argument at the head of the new structure's definition. In other words, the first slots of the new structure are equivalent to (i.e. have the same names as, have the same inits as, etc.) the slots of the argument to the include option. The argument to the include option must be the name of a previously defined structure of the same type as the new one. If no type is specified in the new structure, then it is defaulted to that of the included one. It is an error for the include option to be present without an argument. Note that include does not work on certain types of structures (e.g. structures of type tree or list*). Note also that the conc-name, default-pointer, but-first and callable-accessors options only apply to the accessors defined in the current defstruct; no new accessors are defined for the included slots.

An example:

```
(defstruct (person (type list)
                   conc-name)
  name
  age
  sex)

(defstruct (spaceman (include person)
                     default-pointer)
  helmet-size
  (favorite-beverage 'tang))
```

Now we can make a spaceman like this:

```
(setq spaceman (make-spaceman name 'buzz
                              age 45.
                              sex t
                              helmet-size 17.5))
```

To find out interesting things about spacemen:

```
(helmet-size)           ==> (cadddr spaceman)
(person-name spaceman)  ==> (car spaceman)
(favorite-beverage x)   ==> (car (cddddr x))
```

As you can see the accessors defined for the person structure have names that start with "person-" and they only take one argument. The names of the accessors for the last two slots of the spaceman structure are the same as the slot names, but they allow their argument to be omitted. The accessors for the first three slots of the spaceman structure are the same as the accessors for the person structure.

Often, when one structure includes another, the default initial values supplied by the included structure will be undesirable. These default initial values can be modified at the time of inclusion by giving the include option as:

`(include` *name new-init-1 ... new-init-n*`)`

Each *new-init-i* is either the name of an included slot or of the form (*included-slot-name new-init*). If it is just a slot name, then in the new structure (the one doing the including) that slot will have no initial value. If a new initial value is given, then that code replaces the old initial value code for that slot in the new structure. The included structure is unmodified.

9.4.7 named

This option tells defstruct that you desire your structure to be a "named structure". On PDP-10s this means you want your structure implemented with a named-hunk or named-list. On a Lisp Machine this indicates that you desire either a named-array or a named-array-leader or a named-list. On Multics this indicates that you desire a named-list. defstruct bases its decision as to what named type to use on whatever value you did or didn't give to the type option.

It is an error to use this option with an argument.

9.4.8 make-array

Available only on Lisp Machines, this option allows the user to control those aspects of the array used to implement the structure that are not otherwise constrained by defstruct (such as the area it is to be allocated in).

The argument to the make-array option should be a list of alternating keyword symbols to the Lisp Machine make-array function (see the Lisp Machine manual), and forms whose values are to be the arguments to those keywords. For example, (make-array (:type 'art-4b)) would request that the type of the array be art-4b. Note that the keyword symbols are *not* evaluated.

Constructor macros for structures implemented as arrays all allow the keyword make-array to be supplied. Its argument is of the same form as the make-array option, and attributes specified there (in the constructor form) will override those given in the defstruct form.

Since it is sometimes necessary to be able to specify the dimensions of the array that defstruct is going to construct (for structures of type array-leader for example), the make-array option or constructor keyword accepts the additional keywords :length and :dimension (they mean the same thing). The argument to this pseudo make-array keyword will be supplied as the first argument to the make-array function when the constructor is expanded.

defstruct chooses appropriate defaults for those attributes not specified in the defstruct form or in the constructor form, and defstruct overrides any specified attributes that it has to.

9.4.9 sfa-function

Available only on PDP-10s, this option allows the user to specify the function that will be used in structures of type sfa. Its argument should be a piece of code that evaluates to the desired function. Constructor macros for this type of structure will take sfa-function as a keyword whose argument is also the code to evaluate to get the function, overriding any supplied in the original defstruct form.

If sfa-function is not present anywhere, then the constructor will use the name-symbol of the structure as the function.

9.4.10 sfa-name

Available only on PDP-10s, this option allows the user to specify the object that will be used in the printed representation of structures of type sfa. Its argument should be a piece of code that evaluates to that object. Constructor macros for this type of structure will take sfa-name as a keyword whose argument is also the code to evaluate to get the object to use, overriding any supplied in the original defstruct form.

If sfa-name is not present anywhere, then the constructor will use the name-symbol of the structure as the function.

9.4.11 external-ptr

Available only on Multics, this option is used with structures of type external. Its argument should be a piece of code that evaluates to a fixnum "packed pointer" pointing to the first word of the external array the defstruct is to construct. Constructor macros for this type of structure will take external-ptr as a keyword whose argument overrides any supplied in the original defstruct form.

If external-ptr is not present anywhere, then the constructor signals an error when it expands.

9.4.12 size-symbol

The size-symbol option allows a user to specify a symbol whose value will be the "size" of the structure. The exact meaning of this varies, but in general this number is the one you would need to know if you were going to allocate one of these structures yourself. The symbol will have this value both at compile time and at run time. If this option is present without an argument, then the name of the structure is concatenated with "-size" to produce the symbol.

9.4.13 size-macro

Similar to size-symbol. A macro of no arguments is defined that expands into the size of the structure. The name of this macro defaults as with size-symbol.

9.4.14 initial-offset

This option allows you to tell defstruct to skip over a certain number of slots before it starts allocating the slots described in the body. This option requires an argument, which must be a fixnum, which is the number of slots you want defstruct to skip. To make use of this option requires that you have some familiarity with how defstruct is implementing you structure, otherwise you will be unable to make use of the slots that defstruct has left unused.

9.4.15 but-first

This option is best explained by example:

```
(defstruct (head (type list)
                 (default-pointer person)
                 (but-first person-head))
  nose
  mouth
  eyes)
```

So now the accessors expand like this:

```
(nose x)        ==> (car (person-head x))
(nose)          ==> (car (person-head person))
```

The theory is that but-first's argument will likely be an accessor from some other structure, and it is never expected that this structure will be found outside of that slot of that other structure. (In the example I had in mind that there was a person structure which had a slot accessed by person-head.) It is an error for the but-first option to be used without an argument.

9.4.16 callable-accessors

This option controls whether the accessors defined by defstruct will work as "functional arguments". (As the first argument to mapcar, for example.) On the Lisp Machine accessors are callable by default, but on PDP-10s it is expensive to make this work, so they are only callable if you ask for it. (Currently on Multics the feature doesn't work at all...) The argument to this option is nil to indicate that the feature should be turned off, and t to turn the feature on. If the option is present with no argument, then the feature is turned on.

9.4.17 eval-when

Normally the macros defined by defstruct are defined at eval-time, compile-time and at load-time. This option allows the user to control this behavior. (eval-when (eval compile)), for example, will cause the macros to be defined only when the code is running interpreted and inside the compiler, no trace of defstruct will be found when running compiled code.

Using the eval-when option is preferable to wrapping an eval-when around a defstruct form, since nested eval-whens can interact in unexpected ways.

9.4.18 property

For each structure defined by defstruct, a property list is maintained for the recording of arbitrary properties about that structure.

The property option can be used to give a defstruct an arbitrary property. (property *property-name value*) gives the defstruct a *property-name* property of *value*. Neither argument is evaluated. To access the property list, the user will have to look inside the defstruct-description structure himself, he is referred to section 9.7, page 41, for more information.

9.4.19 A Type Used As An Option

In addition to the options listed above, any currently defined type (a legal argument to the type option) can be used as a option. This is mostly for compatibility with the old Lisp Machine defstruct. It allows you to say just *type* when you should be saying (type *type*). Use of this feature in new code is discouraged. It is an error to give an argument to a type used as an option in this manner.

9.4.20 Other Options

Finally, if an option isn't found among those listed above, defstruct checks the property list of the name of the option to see if it has a non-null defstruct-option property. If is does have such a property, then if the option was of the form (*option-name value*), it is treated just like (property *option-name value*). That is, the defstruct is given an *option-name* property of *value*. It is an error to use such an option without a value.

This provides a primitive way for the user to define his own options to defstruct. Several of the options listed above are actually implemented using this mechanism.

9.5 Byte Fields

On Multics, the byte field feature will not work unless the user has arranged to define the functions ldb and dpb (section 8.1.2, page 18). They are not yet present in the default environment, but they are available as part of the extension library (section 11.2, page 59).

The byte field feature of defstruct allows the user to specify that several slots of his structure are bytes in a fixed point number stored in one element of the structure. For example, suppose we had the following structure:

```
(defstruct (phone-book-entry (type list))
  name
  address
  (area-code 617.)
  exchange
  line-number)
```

This will work just fine. Except you notice that an area-code and an exchange are both always less than 1000., and so both can easily fit in 10. bits, and the line-number is always less than 10000. and can thus fit in 14. bits. Thus you can pack all three parts of a phone number in 34.

bits. If you have a lisp with 36. bit fixnums, then you should be able to put the entire phone number in one fixnum in a structure. defstruct allows you to do this as follows:

```
(defstruct (phone-book-entry (type list))
  name
  address
  ((area-code 3012 617.)
   (exchange 1612)
   (line-number 0016)))
```

The magic numbers 3012, 1612 and 0016 are byte specifiers suitable for use with the functions ldb and dpb (page 18). Things will expand as follows:

```
(area-code pbe) ==> (ldb 3012 (caddr pbe))
(exchange pbe)  ==> (ldb 1612 (caddr pbe))

(make-phone-book-entry
        name '|Fred Derf|
        address '|259 Octal St.|
        exchange ex
        line-number 7788.)

==> (list '|Fred Derf| '|259 Octal St.| (dpb ex 1612 115100017154))

(alter-phone-book-entry pbe
                        exchange ex
                        line-number ln)

==> ((lambda (G0003)
       (setf (caddr G0003)
             (dpb ex 1612 (dpb ln 0016 (caddr G0003)))))
     pbe)
```

defstruct tries to be maximally clever about constructing and altering structures with byte fields.

The byte specifiers are actually pieces of code that are expected to evaluate to byte specifiers, but defstruct will try and understand fixnums if you supply them. (In the make-phone-book example, defstruct was able to make use of its knowledge of the line-number and area-code byte specifiers to assemble the constant number 115100017154 and produce code to just deposit in the exchange.)

A nil in the place of the byte specifier code means to define an accessor for the entire word. So we could say:

```
(defstruct (phone-book-entry (type list))
  name
  address
  ((phone-number nil)
   (area-code 3012 617.)
   (exchange 1612)
   (line-number 0016)))
```

to enable us to do things like:

```
(setf (phone-number pbe1) (phone-number pbe2))
```

to cause two entries to have the same phone numbers.

We could also have said just: ((phone-number) ...) in that last defstruct, but the feature of nil byte specifiers allows you to supply initial values for the entire slot by saying: ((*name* nil *init*) ...).

Constructor macros initialize words divided into byte fields as if they were deposited in the following order:

1) Initializations for the entire word given in the defstruct form.

2) Initializations for the byte fields given in the defstruct form.

3) Initializations for the entire word given in the constructor macro form.

4) Initializations for the byte fields given in the constructor macro form.

Alterant macros operate in a similar manner. That is, as if the entire word was modified first, and then the byte fields were deposited. Results will be unpredictable in constructing and altering if byte fields that overlap are given.

9.6 About Autoloading

This section only applies to PDP-10 and Multics Lisp.

If you look at the property lists of the macros defined by defstruct, you will find that they are all have macro properties of one of four functions: defstruct-expand-ref-macro, defstruct-expand-cons-macro, defstruct-expand-alter-macro and defstruct-expand-size-macro. These functions figure out how to expand the macro by examining the property list of the car of the form they are asked to expand. defstruct-expand-ref-macro, for example, looks for a defstruct-slot property, which should be a cons of the form (*structure-name . slot-name*).

Since the defstruct form only expands into putprops of the desired functions (instead of actually constructing a full-fledged definition), loading a compiled file containing a defstruct merely adds a few properties to some symbols. The run time environment is not needlessly cluttered with unwanted list structure or subr objects. If the user thinks he may wish to use any of the macros defined by defstruct after compiling his file, he need only give the four expanding functions autoload properties of the name of the file containing defstruct itself.

For purposes of using defstruct interpreted, the two symbols defstruct and defstruct-define-type should be given similar autoload properties. Thus six symbols with autoload properties suffice to make defstruct appear loaded at all times.

9.7 The defstruct-description Structure

This section discusses the internal structures used by defstruct that might be useful to programs that want to interface to defstruct nicely. The information in this section is also necessary for anyone who is thinking of defining his own structure types (section 9.8, page 42). Lisp Machine programmers will find that the symbols found only in this section are all interned in the "systems-internals" package.

Whenever the user defines a new structure using defstruct, defstruct creates an instance of the defstruct-description structure. This structure can be found as the defstruct-description property of the name of the structure; it contains such useful information as the name of the structure, the number of slots in the structure, etc.

The defstruct-description structure is defined something like this: (This is a bowdlerized version of the real thing, I have left out a lot of things you don't need to know unless you are actually reading the code.)

```
(defstruct (defstruct-description
             (default-pointer description)
             (conc-name defstruct-description-))
  name
  size
  property-alist
  slot-alist)
```

The name slot contains the symbol supplied by the user to be the name of his structure, something like spaceship or phone-book-entry.

The size slot contains the total number of slots in an instance of this kind of structure. This is *not* the same number as that obtained from the size-symbol or size-macro options to defstruct. A named structure, for example, usually uses up an extra location to store the name of the structure, so the size-macro option will get a number one larger than that stored in the defstruct description.

The property-alist slot contains an alist with pairs of the form (*property-name . property*) containing properties placed there by the property option to defstruct or by property names used as options to defstruct (see section 9.4.18, page 38, and section 9.4.20, page 38).

The slot-alist slot contains an alist of pairs of the form (*slot-name . slot-description*). A *slot-description* is an instance of the defstruct-slot-description structure. The defstruct-slot-description structure is defined something like this: (another bowdlerized defstruct)

```
(defstruct (defstruct-slot-description
             (default-pointer slot-description)
             (conc-name defstruct-slot-description-))
  number
  ppss
  init-code
  ref-macro-name)
```

The number slot contains the number of the location of this slot in an instance of the structure. Locations are numbered starting with 0, and continuing up to one less than the size of the structure. The actual location of the slot is determined by the reference consing code associated with the type of the structure. See section 9.8, page 42.

The ppss slot contains the byte specifier code for this slot if this slot is a byte field of its location. If this slot is the entire location, then the ppss slot contains nil.

The init-code slot contains the initialization code supplied for this slot by the user in his defstruct form. If there is no initialization code for this slot then the init-code slot contains the symbol %%defstruct-empty%%.

The ref-macro-name slot contains the symbol that is defined as a macro that expands into a reference to this slot.

9.8 Extensions to defstruct

defstruct-define-type *Macro*

The macro defstruct-define-type can be used to teach defstruct about new types it can use to implement structures.

9.8.1 A Simple Example

Let us start by examining a sample call to defstruct-define-type. This is how the list type of structure might have been defined:

```
(defstruct-define-type list
  (cons (initialization-list description keyword-options) list
        (cons 'list initialization-list))
  (ref (slot-number description argument)
       (list 'nth slot-number argument)))
```

This is the minimal example. We have provided defstruct with two pieces of code, one for consing up forms to construct instances of the structure, the other to cons up forms to reference various elements of the structure.

From the example we can see that the constructor consing code is going to be run in an environment where the variable initialization-list is bound to a list which is the initializations to the slots of the structure arranged in order. The variable description will be bound to the defstruct-description structure for the structure we are consing a constructor for. (See section 9.7, page 41.) The binding of the variable keyword-options will be described later. Also the

symbol list appears after the argument list, this conveys some information to defstruct about how the constructor consing code wants to get called.

The reference consing code gets run with the variable slot-number bound to the number of the slot that is to be referenced and the variable argument bound to the code that appeared as the argument to the accessor macro. The variable description is again bound to the appropriate instance of the defstruct-description structure.

This simple example probably tells you enough to be able to go ahead and implement other structure types, but more details follow.

9.8.2 Syntax of defstruct-define-type

The syntax of defstruct-define-type is

```
(defstruct-define-type type
  option-1
  ...
  option-n)
```

where each *option-i* is either the symbolic name of an option or a list of the form (*option-i . rest*). (Actually *option-i* is the same as (*option-i*).) Different options interpret *rest* in different ways.

The symbol *type* is given a defstruct-type-description property of a structure that describes the type completely.

9.8.3 Options to defstruct-define-type

This section is a catalog of all the options currently known about by defstruct-define-type.

9.8.3.1 cons

The cons option to defstruct-define-type is how the user supplies defstruct with the necessary code that it needs to cons up a form that will construct an instance of a structure of this type.

The cons option has the syntax:

```
(cons (inits description keywords) kind
      body)
```

body is some code that should construct and return a piece of code that will construct, initialize and return an instance of a structure of this type.

The symbol *inits* will be bound to the code that the constructor conser should use to initialize the slots of the structure. The exact form of this argument is determined by the symbol *kind*. There are currently two kinds of initialization. There is the list kind, where *inits* is bound to a list of initializations, in the correct order, with nils in uninitialized slots. And there is the alist

kind, where *inits* is bound to an alist with pairs of the form (*slot-number* . *init-code*).

The symbol *description* will be bound to the instance of the defstruct-description structure (section 9.7, page 41) that defstruct maintains for this particular structure. This is so that the constructor conser can find out such things as the total size of the structure it is supposed to create.

The symbol *keywords* will be bound to a alist with pairs of the form (*keyword* . *value*), where each *keyword* was a keyword supplied to the constructor macro that wasn't the name of a slot, and *value* was the "code" that followed the keyword. (See section 9.8.3.5, page 45, and section 9.4.2, page 30.)

It is an error not to supply the cons option to defstruct-define-type.

9.8.3.2 ref

The ref option to defstruct-define-type is how the user supplies defstruct with the necessary code that it needs to cons up a form that will reference an instance of a structure of this type.

The ref option has the syntax:

```
(ref (number description arg-1 ... arg-n)
     body)
```

body is some code that should construct and return a piece of code that will reference an instance of a structure of this type.

The symbol *number* will be bound to the location of the slot that the is to be referenced. This is the same number that is found in the number slot of the defstruct-slot-description structure (section 9.7, page 41).

The symbol *description* will be bound to the instance of the defstruct-description structure that defstruct maintains for this particular structure.

The symbols *arg-i* are bound to the forms supplied to the accessor as arguments. Normally there should be only one of these. The *last* argument is the one that will be defaulted by the default-pointer option (section 9.4.4, page 32). defstruct will check that the user has supplied exactly n arguments to the accessor macro before calling the reference consing code.

It is an error not to supply the ref option to defstruct-define-type.

9.8.3.3 overhead

The overhead option to defstruct-define-type is how the user declares to defstruct that the implementation of this particular type of structure "uses up" some number of slots locations in the object actually constructed. This option is used by various "named" types of structures that store the name of the structure in one location.

The syntax of overhead is:

```
(overhead n)
```

where *n* is a fixnum that says how many locations of overhead this type needs.

This number is only used by the size-macro and size-symbol options to defstruct. (See section 9.4.13, page 36, and section 9.4.12, page 36.)

9.8.3.4 named

The named option to defstruct-define-type controls the use of the named option to defstruct. With no argument the named option means that this type is an acceptable "named structure". With an argument, as in (named *type-name*), the symbol *type-name* should be that name of some other structure type that defstruct should use if someone asks for the named version of this type. (For example, in the definition of the list type the named option is used like this: (named named-list).)

9.8.3.5 keywords

The keywords option to defstruct-define-type allows the user to define constructor keywords (section 9.4.2, page 30) for this type of structure. (For example the make-array constructor keyword for structures of type array on Lisp Machines.) The syntax is:

```
(keywords keyword-1 ... keyword-n)
```

where each *keyword-i* is a symbol that the constructor conser expects to find in the *keywords* alist (section 9.8.3.1, page 43).

9.8.3.6 defstruct

The defstruct option to defstruct-define-type allows the user to run some code and return some forms as part of the expansion of the defstruct macro.

The defstruct option has the syntax:

```
(defstruct (description)
           body)
```

body is a piece of code that will be run whenever defstruct is expanding a defstruct form that defines a structure of this type. The symbol *description* will be bound to the instance of the defstruct-description structure that defstruct maintains for this particular structure.

The value returned by the **defstruct** option should be a *list* of forms to be included with those that the defstruct expands into. Thus, if you only want to run some code at defstruct expand time, and you don't want to actually output any additional code, then you should be careful to return nil from the code in this option.

10. Format

format *destination control-string* (any-number-of *args*)

format is used to produce formatted output. format outputs the characters of *control-string*, except that tilde ("~") introduces a directive. The character after the tilde, possibly preceded by arguments and modifiers, specifies what kind of formatting is desired. Some directives use an element of *args* to create their output.

The output is sent to *destination*. If *destination* is nil, a string is created which contains the output (see section 10.4 on format and strings, page 57). If *destination* is t, the output is sent to the "default output destination", which in Maclisp is the output filespec nil—the terminal (controlled by the variable ^w) and outfiles (controlled by ^r). With those exceptions, *destination* may be any legitimate output file specification.

A directive consists of a tilde, optional decimal numeric parameters separated by commas, optional colon (":") and atsign ("@") modifiers, and a single character indicating what kind of directive this is. The alphabetic case of the character is ignored. Examples of control strings:

```
"~S"        ; This is an S directive with no parameters.
"~3,4:@s"   ; This is an S directive with two parameters, 3 and 4,
            ;   and both the colon and atsign flags.
```

format includes some extremely complicated and specialized features. It is not necessary to understand all or even most of its features to use format efficiently. The beginner should skip over anything in the following documentation that is not immediately useful or clear. The more sophisticated features are there for the convenience of programs with complicated formatting requirements.

Sometimes a numeric parameter is used to specify a character, for instance the padding character in a right- or left-justifying operation. In this case a single quote (') followed by the desired character may be used as a numeric argument. For example, you can use

```
"~5,'0d"
```

to print a decimal number in five columns with leading zeros (the first two parameters to ~D are the number of columns and the padding character).

In place of a numeric parameter to a directive, you can put the letter v, which takes an argument from *args* as a parameter to the directive. Normally this should be a number but it doesn't really have to be. This feature allows variable column-widths and the like. Also, you can use the character # in place of a parameter; it represents the number of arguments remaining to be processed.

It is possible to have a directive name of more than one character. The name need simply be enclosed in backslashes ("\"); for example,

```
(format t "~\now\" (status daytime))
```

As always, case is ignored here. There is no way to quote a backslash in such a construct. No multi-character operators come with format.

Note that the characters @, #, and \ which are used by format are special to the default Multics input processor, and may need to be quoted accordingly when typed in (normally, with \).

Once upon a time, various strange and wonderful interpretations were made on *control-string* when it was neither a string nor a symbol. Some of these are still supported for compatibility with existing code (if any) which uses them; new code, however, should only use a string or symbol for *control-string*.

This document describes an implementation of format which is currently in use in Maclisp (both PDP-10 and Multics), and is intended to be transported to NIL. It thus is oriented towards the Maclisp dialect of Lisp. The behaviour of format operators should be fairly consistent across Lisp dialects; entries documented here other than format, however, exist only in the Maclisp implementation at this time, although they could be added to other format implementations without difficulty.

10.1 The Operators

Here are the operators.

~A *arg*, any Lisp object, is printed without slashification (like princ). ~*n*A inserts spaces on the right, if necessary, to make the column width at least *n*. ~*mincol*,*colinc*,*minpad*,*padchar*A is the full form of ~A, which allows alcborate control of the padding. The string is padded on the right with at least *minpad* copies of *padchar*; padding characters are then inserted *colinc* characters at a time until the total width is at least *mincol*. The defaults are 0 for *mincol* and *minpad*, 1 for *colinc*, and *space* for *padchar*. The atsign modifier causes the output to be right-justified in the field instead of left-justified. (The same algorithm for calculating how many pad characters to output is used.) The colon modifier causes an *arg* of nil to be output as ().

~S This is identical to ~A except that it uses prin1 instead of princ.

~D Decimal integer output. *arg* is printed as a decimal integer. ~*n*,*m*,*o*D uses a column width of *n*, padding on the left with pad-character *m* (default of space), using the character *o* (default comma) to separate groups of three digits. These commas are only inserted if the : modifier is present. Additionally, if the @ modifier is present, then the sign character will be output unconditionally; normally it is only output if the integer is negative. If *arg* is not an integer, then it is output (using princ) right-justified in a field *n* wide, using a pad-character of *m*, with base decimal and *nopoint bound to t.

~O Octal integer output. Just like ~D.

~P If *arg* is not 1, a lower-case "s" is printed. ("P" is for "plural".) ~:P does the same thing, after backing up an argument (like "~:*", below); it prints a lower-case s if the *last* argument was not 1. ~@P prints "y" if the argument is 1, or "ies" if it is not. ~:@P does the same thing, but backs up first.
Example:

```
(format nil "~D Kitt~:@P" 3)  =>  "3 Kitties"
```

~* ~* ignores one *arg*. ~*n** ignores the next *n* arguments. *n* may be negative. ~:* backs up one arg; ~*n*:* backs up *n* args.

~*n*G "Goes to" the *n*th argument. ~0G goes back to the first argument in *args*. Directives after a ~*n*G will take sequential arguments after the one gone to. Note that this command, and ~*, only affect the "local" *args*, if "control" is within something like ~{.

~% Outputs a newline. ~*n*% outputs *n* newlines. No argument is used.

~& The fresh-line operation is performed on the output stream. ~*n*& outputs *n*-1 newlines after the fresh-line. The fresh-line operation says to do a terpri unless the cursor is at the start of the line. This operation will virtually always succeed in Maclisp, since all Maclisp file arrays know their charpos. Implemented by format-fresh-line, page 56.

~X Outputs a space. ~*n*X outputs *n* spaces. No argument is used.

~~ Outputs a tilde. ~*n*~ outputs *n* tildes. No argument is used.

~*newline*
Tilde immediately followed by a carriage return ignores the carriage return and any whitespace at the beginning of the next line. With a :, the whitespace is left in place. With an @, the carriage return is left in place. This directive is typically used when a format control string is too long to fit nicely into one line of the program:

```
(format the-output-stream "~&This is a reasonably ~
                           long string~%")
```

which is equivalent to formating the string

```
"~&This is a reasonably long string~%"
```

~| Outputs a formfeed. ~*n*| outputs *n* formfeeds. No argument is used. This is implemented by format-formfeed, page 56.

~T Spaces over to a given column. The full form is ~*destination*,*increment*T, which will output sufficient spaces to move the cursor to column *destination*. If the cursor is already past column *destination*, it will output spaces to move it to column *destination* + *increment* * *k*, for the smallest integer value *k* possible. *increment* defaults to 1. This is implemented by the format-tab-to function, page 56.

~Q ~Q uses one argument, and applies it as a function to *params*. It could thus be used to, for example, get a specific printing function interfaced to format without defining a specific operator for that operation, as in

```
(format t "~&;  The frob ~vQ is not known.~%"
        frob 'frob-printer)
```

The printing function should obey the conventions described in section 10.3, page 54. Note that the function to ~Q follows the arguments it will get, because they are passed in as format parameters which get collected before the operator's argument.

~[~[*str0*~;*str1*~;...~;*strn*~] is a set of alternative control strings. The alternatives (called *clauses*) are separated by ~; and the construct is terminated by ~]. For example, "~[Siamese ~;Manx ~;Persian ~;Tortoise-Shell ~;Tiger ~;Yu-Hsiang ~]kitty". The *arg*th alternative is selected; 0 selects the first. If a numeric parameter is given

(i.e. ~*n*[), then the parameter is used instead of an argument (this is useful only if the parameter is "#"). If *arg* is out of range no alternative is selected. After the selected alternative has been processed, the control string continues after the ~].

~[*str0*~;*str1*~;...~;*strn*~:;*default*~] has a default case. If the *last* ~; used to separate clauses is instead ~:;, then the last clause is an "else" clause, which is performed if no other clause is selected. For example, "~[Siamese ~;Manx ~;Persian ~;Tortoise-Shell ~;Tiger ~;Yu-Hsiang ~:;Unknown ~] kitty".

~[~*tag00*,*tag01*,...;*str0*~*tag10*,...;*str1*...~] allows the clauses to have explicit tags. The parameters to each ~; are numeric tags for the clause which follows it. That clause is processed which has a tag matching the argument. If ~:*a1*,*a2*,*b1*,*b2*,...; is used, then the following clause is tagged not by single values but by ranges of values *a1* through *a2* (inclusive), *b1* through *b2*, etc. ~:; with no parameters may be used at the end to denote a default clause. For example, "~[~'+,'-,'*,'//;operator ~'A,'Z,'a,'z;letter ~'0,'9;digit ~:;other ~]".

~:[*false*~;*true*~] selects the *false* control string if *arg* is nil, and selects the *true* control string otherwise.

~@[*true*~] tests the argument. If it is not nil, then the argument is not used up, but is the next one to be processed, and the one clause is processed. If it is nil, then the argument is used up, and the clause is not processed.

```
(setq prinlevel nil prinlength 5)
(format nil "~@[ PRINLEVEL=~D~]~@[ PRINLENGTH=~D]"
        prinlevel prinlength)
  => " PRINLENGTH=5"
```

~R If there is no parameter, then *arg* is printed as a cardinal English number, e.g. four. With the colon modifier, *arg* is printed as an ordinal number, e.g. fourth. With the atsign modifier, *arg* is printed as a Roman numeral, e.g. IV. With both atsign and colon, *arg* is printed as an old Roman numeral, e.g. IIII.

If there is a parameter, then it is the radix in which to print the number. The flags and any remaining parameters are used as for the ~D directive. Indeed, ~D is the same as ~10R. The full form here is therefore ~*radix*,*mincol*,*padchar*,*commachar*R.

~C *arg* is coerced to a character code. With no modifiers, ~C simply outputs this character. ~@C outputs the character so it can be read in again using the # reader macro: if there is a named character for it, that will be used, for example "#\Return"; if not, it will be output like "#/A". ~:C outputs the character in human-readable form, as in "Return", "Meta-A". ~:@C is like ~:C, and additionally might (if warranted and if it is known now) parenthetically state how the character may be typed on the user's keyboard.

To find the name of a character, ~C looks in two places. The first is the value of the symbol which is the value of format:*/#-var, which is initialized to be the variable which the # reader macro uses. It is not necessary for the value of format:*/#-var to be bound. The second place is *format-chnames; this is used primarily to handle non-printing characters, in case the # reader macro is not loaded. Both of these are a-lists, of the form ((*name* . *code*) (*name* . *code*) ...).

The Maclisp/NIL format has no mechanism for telling how a particular character needs to be typed on a keyboard, but it does provide a hook for one. If the value of format:*top-char-printer is not nil, then it will be called as a function on two arguments: the character code, and the character name. If there were bucky-bits present, then they will have been stripped off unless there was a defined name for the character with the bits present. The function should do nothing in normal cases, but if it does it should output two spaces, and then the how-to-type-it-in description in parentheses. See section 10.3, page 54 for information on how to do output within format.

~< *~mincol,colinc,minpad,padchar<text~>* justifies *text* within a field *mincol* wide. *text* may be divided up into segments with ~;—the spacing is evenly divided between the text segments. With no modifiers, the leftmost text segment is left justified in the field, and the rightmost text segment right justified; if there is only one, as a special case, it is right justified. The colon modifier causes spacing to be introduced before the first text segment; the atsign modifier causes spacing to be added after the last. *minpad*, default 0, is the minimum number of *padchar* (default space) padding characters to be output between each segment. If the total width needed to satisfy these constraints is greater than *mincol*, then *mincol* is adjusted upwards in *colinc* increments. *colinc* defaults to 1. For example,

```
(format nil "~10<foo~;bar~>")     => "foo    bar"
(format nil "~10:<foo~;bar~>")    => "  foo  bar"
(format nil "~10:@<foo~;bar~>")   => "  foo bar "
(format nil "~10<foobar~>")       => "    foobar"
(format nil "~10:@<foobar~>")     => "  foobar  "
(format nil "$~10,,,'*<~3f~>" 2.59023)  => "$******2.59"
```

If ~^ is used within a ~< construct, then only the clauses which were completely processed are used. For example,

```
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo)
        => "            FOO"
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo 'bar)
        => "FOO         BAR"
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo 'bar 'baz)
        => "FOO   BAR   BAZ"
```

If the first clause of a ~< is terminated with ~:; instead of ~;, then it is used in a special way. All of the clauses are processed (subject to ~^, of course), but the first one is omitted in performing the spacing and padding. When the padded result has been determined, then if it will fit on the current line of output, it is output, and the text for the first clause is discarded. If, however, the padded text will not fit on the current line, then the text for the first clause is output before the padded text. The first clause ought to contain a carriage return. The first clause is always processed, and so any arguments it refers to will be used; the decision is whether to use the resulting piece of text, not whether to process the first clause. If the ~:; has a numeric parameter *n*, then the padded text must fit on the current line with *n* character positions to spare to avoid outputting the first clause's text. For example, the control string

```
"~%;; ~{~<~%;; ~1:; ~S~>~^,~}.~%"
```

can be used to print a list of items separated by commas, without breaking items over line boundaries, and beginning each line with ";; ". The argument 1 in ~1:; accounts for the width of the comma which will follow the justified item if it is not the last element in the list, or the period if it is. If ~:; has a second numeric parameter, then it is used as the width of the line, thus overriding the natural line width of the output stream. To make the preceding example use a line width of 50, one would write

```
"~%;; ~{~<~%;; ~1,50:; ~S~>~^,~}.~%"
```

Note that the segments ~< breaks the output up into are computed "out of context" (that is, they are first recursively formatted into strings). Thus, it is not a good idea for any of the segments to contain relative-positioning commands (such as ~T and ~&), or any line breaks. If ~:; is used to produce a prefix string, it also should not use relative-positioning commands.

~{*str*~}

This is an iteration construct. The argument should be a list, which is used as a set of arguments as if for a recursive call to format. The string *str* is used repeatedly as the control string. Each iteration can absorb as many elements of the list as it likes. If before any iteration step the list is empty, then the iteration is terminated. Also, if a numeric parameter *n* is given, then there will be at most *n* repetitions of processing of *str*.

~:{*str*~} is similar, but the argument should be a list of sublists. At each repetition step one sublist is used as the set of arguments for processing *str*; on the next repetition a new sublist is used, whether or not all of the last sublist had been processed.

~@{*str*~} is similar to ~{*str*~}, but instead of using one argument which is a list, all the remaining arguments are used as the list of arguments for the iteration.

~:@{*str*~} combines the features of ~:{*str*~} and ~@{*str*~}. All the remaining arguments are used, and each one must be a list. On each iteration one argument is used as a list of arguments.

Terminating the repetition construct with ~:} instead of ~} forces *str* to be processed at least once even if the initial list of arguments is null (however, it will not override an explicit numeric parameter of zero).

If *str* is null, then an argument is used as *str*. It must be a string, and precedes any arguments processed by the iteration. As an example, the following are equivalent:

```
(apply (function format) (list* stream string args))
(format stream "~1{~:}" string args)
```

This will use string as a formatting string. The ~1{ says it will be processed at most once, and the ~:} says it will be processed at least once. Therefore it is processed exactly once, using args as the arguments.

~} Terminates a ~{. It is undefined elsewhere.

~^ This is an escape construct. If there are no more arguments remaining to be processed, then the immediately enclosing ~{ or ~< construct is terminated. (In the latter case, the ~< formatting *is* performed, but no more clauses are processed before doing the justification. The ~^ should appear only at the *beginning* of a ~< clause, because it aborts the entire clause. It may appear anywhere in a ~{ construct.) If there is no such enclosing construct, then the entire formatting operation is terminated.

If a numeric parameter is given, then termination occurs if the parameter is zero. (Hence ~^ is the same as ~#^.) If two parameters are given, termination occurs if they are equal. If three are given, termination occurs if the second is between the other two in ascending order.

If ~^ is used within a ~:{ construct, then it merely terminates the current iteration step (because in the standard case it tests for remaining arguments of the current step only); the next iteration step commences immediately. To terminate the entire iteration process, use ~:^.

~F outputs *arg* in free-format floating-point. ~*n*F outputs *arg* showing at most *n* digits. ~*n*:F will show exactly *n* digits. No other variations are guaranteed at this time; neither is the *exact* interpretation of *n*. It is reasonable to use this, however, when one desires to print a flonum without showing lots of insignificant trailing digits; for example,

```
(format nil "~6f" 259.258995) => "259.259"
```

~E Outputs *arg* in exponential notation; e.g., "2.59259e+2". ~*n*E interprets *n* the same as ~F. No other parameters or flags are guaranteed at this time.

~$ (That's a dollar sign.) ~*rdig*,*ldig*,*field*,*padchar*$ prints *arg*, a flonum, with exactly *rdig* digits after the decimal point (default is 2), at least *ldig* digits preceding the decimal point (default is 1), right justified in a field *field* columns long, padded out with *padchar*. The colon modifier says that we should cause the sign character to be left justified in the field. The atsign modifier says that we should always output the sign character. The *ldig* allows one to specify a portion of the number which does not get zero suppressed.

~\ This is not really an operator. If one desires to use a multi-character format operator, it may be placed within backslashes, as in ~\now\ for the now operator. See page 47.

10.2 Other Entries

?format *destination control-string* (Any-number-of *frobs*)

This is equivalent to format except that *destination* is interpreted like the second argument to print—nil means "the default", and t means "the terminal". This only exists in Maclisp at the moment.

10.3 Defining your own

define-format-op *Macro*

This may be used in two formats:

```
(define-format-op operator varlist body-forms...)
```

and

```
(define-format-op operator fixnum-character-code)
```

The *operator* may be the fixnum code for a character, or a symbol with the same print-name as the operator. Whichever, it is canonicalized (into upper case) and will be interned into the same obarray/package which format resides in. For example, the format operator for *tilde* could be defined as

```
(define-format-op /~ #/~)
```

where "#/~" represents the fixnum character code for tilde.
For the first format, the type of operator is determined by decoding *varlist*, which may have one of the following formats:

(*params-var*)
: An operator of exactly zero arguments; *params-var* will get bound to the parameters list.

(*params-var arg-var*)
: An operator of exactly one argument; *params-var* will get bound to the parameters list, and *arg-var* to the argument.

(*params-var . args-var*)
: An operator of a variable number of args; *params-var* will get bound to the parameters list, and *args-var* to the remaining arguments to format (or to the recursive ~{ arguments). The operator should return as its value some sublist of *args-var*, so that format knows how many were used.

A definition for the appropriate function is produced with a bvl derived from the variables in *varlist* and a body of *body-forms*. (The argument ordering in the function produced is compatible with that on the Lisp Machine, which is *arg-var* (if any) first, and then *params-var*.)

standard-output *Variable*

Output from format operators should be sent to the stream which is the value of standard-output. In the Multics implementation of format, this value may sometimes be an object which is not suitable for being fed to standard Lisp output functions (e.g., princ); format has definitions of various output functions which handle this case properly, and may be used for defining operators which will work compatibly in Multics Maclisp.

They are documented below. Note that because of the way format interprets its destination, it is not necessarily safe to recursively call format on the value of standard-output in PDP-10 Maclisp. It *is* safe, however, to use ?format (page 54) instead, *or* to call format with a *destination* of the symbol format.

Maclisp format will also accept a *destination* of format to mean "use the format destination already in effect". This is primarily for the benefit of Multics Maclisp, since there the value of standard-output cannot be passed around as a stream. The format operator now, which prints the current time, could be defined as

```
(define-format-op now (params)
   params                ; unused
   (let ((now (status daytime)))
     (format 'format "~2,'0D:~2,'0D:~2,'0D"
             (car now) (cadr now) (caddr now))))
```

with the result that

```
(format nil "The current time is ~\now\.")
```

could produce the string

```
"The current time is 02:59:00."
```

format:colon-flag *Variable*
format:atsign-flag *Variable*

These tell whether or not we have seen a colon or atsign respectively while parsing the parameters to a format operator. They are only bound in the toplevel call to format, so are only really valid when the format operator is first called; if the operator does more parameter parsing (like ~[does) their values should be saved if they will be needed.

These variables used to be named just colon-flag and atsign-flag. In the interest of transporting format code to Lisp implementations with packages, their names have been changed. Thus, in either implementation one references them with the "format:" at the front of the name, which in Maclisp is just part of the print-name.

The *params* are passed in as a list. This list, however, is temporary storage only. If it is going to be passed back, it *must be copied*. In Maclisp and NIL, it is an ordinary list which, in PDP-10 Maclisp, will be reclaimed after the operator has run. On the Lisp Machine, it will be a list-pointer into an art-q-list array, possibly in a temporary area. Thus, although it is safe to save values in this list with rplaca, one should not ever use rplacd on it, either explicitly or implicitly (by use of nconc or nreverse).

Conceptually, format operates by performing output to some stream. In practice, this is what occurs in most implementations; in Maclisp, there are a few special SFAs used by format. This may not be possible in all implementations, however. To get around this, format has a mechanism for allowing the output to go to a pseudo-stream, and supplies a set of functions which will interact with these when they are used.

format-tyo *character*
tyos *character* to the format output destination.

format-princ *object*
princs *object* to the format output destination.

format-prin1 *object*
prin1s *frob* to the format output destination.

format-lcprinc *string capitalize?*
This outputs *string*, which must be a string or symbol, to the format output destination in lower-case. If *capitalize?* is not nil, then the first character is converted to upper case rather than lower.

format-terpri
Does a terpri to the format output destination.

format-charpos
format-linel
Return the charpos and linel of the format output destination. Since in the Maclisp implementation multiple output destinations may be implicitly in use (via outfiles, for instance) this attempts to choose a representative one. The terminal is preferred if it is involved.

format-fresh-line
This performs the fresh-line operation to the default format destination. In PDP-10 Maclisp, this first will try the fresh-line operation if the destination is an SFA and supports it. Otherwise, if the destination is a terminal or an SFA which supports cursorpos, it will try (cursorpos 'a). Otherwise, it will do a terpri if the charpos is not 0. In the Maclisp implementation, where multiple output destinations may be implicitly involved (via outfiles, for instance), this handles each such destination separately.

format-tab-to (fixnum *destination*) (Optional *increment?*)
This implements ~T to the current format destination (q.v.). In PDP-10 Maclisp, this operation on an SFA will use the tab-to operation if it supported, passing in arguments of *destination* and *increment* (as a dotted pair); otherwise, charpos will be used to compute the number of spaces to be output. If charpos is not supported, two spaces will be output.

format-formfeed
Performs a formfeed on the format output destination. In Multics Maclisp, this will normally just tyo the character code for a formfeed. In PDP-10 Maclisp, this will use the formfeed operation if the destination is an SFA and supports it, otherwise it will do a (cursorpos 'c) if the destination is a TTY file array (or an SFA) and supports it, otherwise it simply outputs the character code for a formfeed.

format-flatc *Macro*

(format-flatc *form1 form2 ... formn*)

The *forms* are evaluated in an environment similar to that used inside of format: the various format output-performing routines such as format-tyo and format-princ may be used to "perform output". In all but the Multics Maclisp implementation, standard-output will be a stream which simply counts the characters output—it will only support the tyo operation.

10.4 Format and Strings

In the PDP-10 Maclisp implementation, format has provision for using a user supplied string implementation. Normally, format expects to use symbols. However, if (fboundp 'stringp) is true, then format will use the stringp predicate to see if its argument is a string. If that is the case, then the function string-length will be used to find the size of the string, and char-n will be used to fetch characters out of the string. Both of these routines should have been declared fixnum when compiled (i.e., be ncallable). Internally, tests are ordered such that string-ness is independent on atomic-ness. In addition, the character routine may be used to canonicalize something to a character code.

The Multics implementation is similar to the PDP-10 Maclisp implementation, but uses different routines; stringlength to get the size of the string (or symbol), and getcharn to fetch a character out of the string. The character routine is not used.

***format-string-generator** *Variable*

This variable, which exists only in the Maclisp implementation of format, should have as its value a function to convert a list of characters to a "string" to be returned by format. In the PDP-10 implementation, this defaults to maknam, but may be modified if "strings" are being supported. In the Multics implementation, it is a function which does

(get_pname (maknam *character-list*))

and may be modified, if desired, to something more efficient. In the PDP-10 implementation, the list of characters should neither be modified nor returned to free storage, as it will be reclaimed.

The PDP-10 Maclisp hack of returning an uninterned symbol which has itself as its value and a +internal-string-marker property is not handled here; it is done by the outer call to format itself, and only if the returned "string" is a symbol and the value of *format-string-generator is maknam. This is done so as to not add unnecessary overhead to internal uses of "strings" by format.

The name of this variable differs from that of other user-accessible format variables for historical reasons; it will not be changed, because it only exists in Maclisp.

11. System Differences

This chapter describes differences you may encounter in using these tools in each of the various Lisp dialects in which they have been implemented. One section is devoted to each implementation, and a final section deals with transporting code between them. The system-specific sections are broken into parallel subsections.

Since not all of the tools documented herein will be a part of the default Lisp environment, the first subsection simply describes how to make them available. This will in general involve placing a form at the head of a source file to establish the appropriate read-time and compile-time environment.

The next subsection lists a number of things to watch out for in using a particular implementation or in writing transportable code. It deals with miscellaneous incompatibilities related to these tools and to the Lisp implementations in general. Some options which are specific only to a single implementation are documented here.

The final subsection contains references to other sources of documentation, including that which is available online.

11.1 PDP-10

PDP-10 Maclisp is currently in a state of flux with regard to how these tools are provided and exactly where they are located. Some are present in the default environment while others must be requested explicitly. Check the online documentation for the current status.

11.1.1 Where To Find It

The sharpsign and backquote reader macros are present in the default environment. **loop** and **format** have autoload properties. Many of the functions and special forms described in chapter 8 are present natively or are autoloaded from `((LISP) MLMAC)` (for MacLisp MACros). The rest may be loaded from `((LISP) UMLMAC)` (for User MacLisp MACros). **defstruct** may be loaded from `((LISP) STRUCT)`.

To use the **bit-test**, **dolist**, and **dotimes** macros, place the following form at the head of the source file.

```
(eval-when (eval compile) (load '((lisp) umlmac)))
```

To use **defstruct**, include the following form.

```
(eval-when (eval compile) (load '((lisp) struct)))
```

This will cause **defstruct** to be present during the interpretation or compilation of a file. To use defstruct during debugging of the compiled file, see section 9.6, page 40.

11.1.2 Things To Watch Out For

defun&-check-args *Variable*

The "extended defun" facility (page 8) provides little or no argument count checking for functions by default. By setting this variable to t, the function being defined will contain additional code which will provide a more meaningful error message when the function is called with the incorrect number of arguments.

A feature is provided whereby sequences of characters surrounded by balanced double-quotes are read as un-interned symbols which are bound to themselves. This provides partial compatibility with newer Lisps that have strings. They are primarily useful as arguments to princ, load, and format, and are not intended to be used as first-class data objects as on those systems which support them natively.

11.1.3 Further Documentation

For the latest changes to this implementation, see the file .INFO.;LISP RECENT on any ITS system. Earlier editions of this file are archived in .INFO.;LISP NEWS. The file .INFO.;LISP FORMAT contains a chart of the format operators suitable for printing on an ascii console. The files .INFO.;LISP LOOP and LIBDOC;STRUCT > contain the Bolio source for the loop memo and the defstruct portion of this memo. Perhaps someday these will be replaced by something formatted for a console.

11.2 Multics

The Multics implementation is also changing. As of this writing, only some of the extensions described in this document are available from the standard libraries, but we expect the remainder to be installed in the near future. Check the online documentation for the current status.

11.2.1 Where To Find It

Only a few of the improvements to Multics Maclisp since 1974 are now a part of the default environment. Primarily, these are the special forms which need to be primitively understood by the compiler, such as eval-when and unwind-protect and certain simple functions such as list*. The special forms let and let* are also in the default environment. The other tools documented here may be accessed by the Multics Lisp special form %include. This form causes a text file to be inserted inline during the interpretation or compilation of a file. The form:

```
(%include library)
```

can be placed at the front of any file of Lisp code that wants to utilize all of the features documented here. This form will arrange for the correct eval-time, compile-time and run-time environments to be present whenever the file is being processed in any way. To arrange for this extended environment to be present whenever the lisp interpreter is being used, this form may be placed in the file start_up.lisp in the user's home directory.

Since the %include form is unique to the Multics implementation, a variant on the following may be used to allow the file to also read into other Lisps:

```
(eval-when (eval compile) (or (status feature Multics) (read)))
(%include library)
```

Those Multics Lisp users who wish to be more selective about the facilities they use may instead use the form

```
(%include module)
```

where *module* is one of backquote, sharpsign, defun, defmacro, defstruct, setf, format, or loop. Selective loading of packages may be desired to prevent name or syntax clashes or to speed compilation. Note that some packages will load others as needed. For instance, defstruct will load setf.

%include uses the translator search list to find the file to be included. To see the full pathname of the file which is found, type

```
where_search_paths translator backquote.incl.lisp
```

The actual object segments are bound together as `bound_lisp_library_`.

```
where bound_lisp_library_
```

will find the full pathname of this segment.

The modules listed above may be broken into three categories: read-time (backquote, sharpsign), compile-time (defun, defmacro, setf, defstruct, loop), and run-time (format).

The behavior of the include file for each module depends upon its type. For read-time and compile-time files, the include file will load the file at eval-time or compile-time, but will not add any forms to the object segment. For run-time files, the include file will place a form in the object segment which will load the desired module, either directly or via an autoload property. It will also provide the appropriate function declarations for the compiler.

To use an eval-time or compile-time module at run-time, you can type (%include *module*) to the interpreter or place this form in a file to be read into the interpreter, such as the `start_up.lisp` file. Alternately, you can load the object segment directly, as in (load ">exl>object>lisp_backquote_"), but this is not recommended since it requires specifying an absolute pathname.

11.2.2 Things To Watch Out For

The characters sharpsign ("#") and atsign ("@") are default erase and kill characters on Multics. If these characters are being used for input editing, you will have to type "\#" or "\@" to enter them. Likewise, remember that to directly enter a backslash, two must be typed.

Most other Lisp readers translate lowercase characters to uppercase characters in symbol names. The Multics implementation does not do this case translation by default. This form will modify the readtable to correctly read files which are written in uppercase:

```
(do ((i #/a (1+ i)))
    ((> i #/z))
    (setsyntax (- i #o40)
               (boole 7 (apply 'status (list 'syntax i)) #o500)
               i))
```

The syntax used for reading strings is also different from that used elsewhere. In other Lisps, the / character will quote the next character, so /" will insert a double quote character into a string. In Multics Lisp, the / character loses its special meaning and is interpreted as an ordinary alphabetic. To insert a double quote character into a string, the character is typed twice, following the Multics system convention. This incompatibility arose since the implementation of strings in Multics Lisp predated their implementation elsewhere.

While no installed facility is available at the moment for resolving these syntax differences, the authors have a private reader which is compatible with the PDP-10 case and string syntax. Contact one of them for more information.

When the Multics Lisp compiler needs to generate an anonymous function, it creates a symbol to put the definition on. This will occur whenever a function is passed as an argument using (function (lambda ...)), or when using (defun (*name prop*) ...), for example. Unfortunately, you get the same names every time you run the compiler. Doing

```
(declare (genprefix unique-name))
```

will fix this problem; the compiler will then use *unique-name* as a basis for its generated names. For example, the loop module does

```
(declare (genprefix loop-iteration/|-))
```

so that the compiler will generate names loop-iteration/|-1, loop-iteration/|-2, etc.

error works incompatibly. The second argument is output following the first, rather than before, as is done elsewhere. It is recommended that you use ferror instead, or define your own error signalling primitive. This is often a good thing to do anyway.

The default setting of the *rset switch is nil. You may find it helpful to turn it on in your start_up.lisp.

If you find a symbol which has become mysteriously unbound, chances are that you have taken the car of a symbol or bignum someplace. The object returned by such an operation is the special marker stored in unbound value cells.

The recently written Multics command display_lisp_object_segment (short name dlos) may be used to examine the contents of compiled Lisp object segments. It is quite useful in verifying the proper execution of complex macros and compile-time facilities.

11.2.3 Further Documentation

Online Lisp documentation resides in the directories `>exl>info` and `>doc>info`. The info segment `lisp.changes.info` describes the latest changes to the Multics implementation. `lisp_manual_update.info` describes earlier changes. A collection of segments `lisp_`*module*`.info`, where *module* is as above, repeat the documentation contained in this manual. Finally, the segment `display_lisp_object_segment.info` describes the `display_lisp_object_segment` command.

These segments may be perused by means of the `help` command. For instance, type "`help lisp.changes`" to view the first of these segments.

11.3 Lisp Machine

On the Lisp Machine, everything described in this document is a part of the default environment. No changes need be made to source files.

Further documentation may be found by consulting the Lisp Machine Manual, the `LMMAN` directory on the `AI` machine, and finally the source code itself. The Zmacs command Meta-period will prompt for a function or variable name and read the source file in which it is defined into a buffer.

11.4 Hints On Writing Transportable Code

This section contains some hard-knocks knowledge gathered by the authors over many tea-filled nights of grief. While we have done our best to distill some coherent advice from our experience, there are no easy answers. This is at times a black art.

No doubt there are techniques (and pitfalls!) which we have overlooked. If you have something which could be added to this section, the authors would like to hear from you.

11.4.1 Conditionalization

Ultimately, despite everyone's best efforts, you are likely to find that your code must be conditionalized in some manner. In this eventuality there are a couple of things to be aware of.

The sharpsign reader macro (chapter 3, page 5) is a very handy tool for conditionalizing code for different sites. However, its indiscriminant use can result in *highly* unreadable code. Frequently, when it seems that conditionalizations are going to need to be sprinkled throughout a piece of code, it is possible to identify a common pattern between them, and replace them with an appropriately defined macro. This macro will have a definition that will be conditionalized for each site that the code runs, and will serve to localize the ugly implementation dependent details. Sometimes this operation actually *improves* the readability of the code, since it forces the programmer to give a name to a pattern present in many places.

As an example, the following macro provides a system-independent way of determining the screen size of a console stream.

```
(defmacro screen-size (stream)
  #+ITS   '(status ttysize ,stream)
  #+Lispm '(multiple-value-bind (width height)
               (funcall ,stream ':size-in-characters)
             (cons height width))
  #-(or ITS Lispm) ''(80. . 24.))
```

Another problem with using any of the conditionalization features of the sharpsign reader macro is the fact that although something like

#+NIL form

does cause the form *form* to be ignored in Lisps that aren't of the NIL variety, it is nevertheless necessary that *form* be readable in those other Lisps. In other words, if *form* contains the use of a reader syntax that is *only* supported in NIL, then it won't work to conditionalize *form* in this manner, because other Lisps are going to have to parse it.

Currently, a frequent cause of such problems is the use of a special character name after #\ that isn't universally understood.

In some situations, large portions of a program will need to be written differently from system to system. Often such portions will deal with issues of operating system interface, such as console or file i/o. In such cases, it is best to define a common interface to this portion, so that this code may be factored out into separate files.

11.4.2 Odds and Ends

Avoid directly inserting into your code constants which are specific to the byte, word, or pointer size of a machine. For instance, use (rot 1 -1) instead of 1_43 to reference the most negative fixnum on a PDP-10. Similarly, use (lsh -1 -1) for the most positive fixnum and (haulong (rot 1 -1)) for the number of bits in a fixnum.

There is only one reliable way to define a function that ignores one or more of its arguments without complaint from the compiler:

```
(defun ignore-second-arg (first second third)
  second        ;ignored
  (list first third))
```

Other conventions do not work universally.

Not all Lisps have strings. However, in most, text surrounded by doublequotes will read in as some kind of object which will print out again in a readable format. This object is suitable for passing to functions such as princ and format, but cannot be universally guaranteed to behave reasonably with functions such as equal.

In Maclisp, the default syntax of the colon character is alphabetic, but it has special meaning on the Lisp Machine. Don't use it in the name of a symbol unless you know what you are doing.

If colons are being used only for denoting keywords, then it is useful to give colon the syntax of whitespace outside the Lisp Machine. This can be accomplished with this Maclisp form:

```
(setsyntax '|:| '| | nil)
```

Don't leave control-V's (circle-plus on the Lisp Machine) lying around randomly, like in valret strings. They have special syntactic meaning on the Lisp Machine.

All PDP-10 Maclisp compiled output ("FASL") files use the same format. It is therefore possible to transport the compiled file between PDP-10s (e.g., from an ITS to a TOPS-20), if the code contained therein is not conditionalized on those differences. The source code for loop, for example, does not contain any #+ or #- conditionalizations which distinguish between any PDP-10 implementations; the FASL file for loop used on TOPS-20 and TOPS-10 sites is the same one used on ITS.

Index