REFERENCE MANUAL

930 LISP

L. Peter Deutsch
Butler W. Lampson

University of California, Berkeley

Document No. 30.50.40

Issued June 5, 1965

Revised February 17, 1966

Office of Secretary of Defense
Advanced Research Projects Agency
Washington 25, D. C.

TABLE OF CONTENTS

1.0 Lisp Data

Lisp operates on data called S-expressions. The basic components of the language are called atoms. An atom may be

1) A name, which is a string of letters and digits of arbitrary length, beginning with a letter. The particular letters and digits used have no significance except to distinguish the name from other names.

2) A number, written as a decimal integer of less than 24 bits, possibly with a negative sign.

3) The special symbols T and NIL. T stands for truth. NIL stands for falsity and a number of other things.

S-expressions are made up of dotted pairs. The simplest dotted pair is a dotted pair of two atoms: (A.B). However, the components of a dotted pair may themselves be dotted pairs: (A.(C.B)), for instance.

The most common form for a dotted pair is a standard representation of a list or ordered set of dotted pairs or atoms. In this representation the ordered set A,B,C,D,E would be written (A.(B.(C.(D.(E.NIL))))). This is the reader's first introduction to Lisp parentheses. Since the explicit notation is somewhat cumbersome, a compressed form is normally used, in which the above list is written (A B C D E). Blanks separate the elements of the list. Any number of blanks may be used instead of one. Although the dotted pair notation is rarely seen outside of the introductory section of a manual, it is the basis of Lisp data and program structure and can always be referred back to in case of confusion.

2.0 Lisp programs

A Lisp program is basically an S-expression which is interpreted as a function call according to the following rule: a list is evaluated by taking its first element as a function name and subsequent elements are arguments of the function. All arguments are evaluated before the function is called. Arguments may themselves be lists, i.e. function calls. Thus (A B C) is a call of the function A with arguments obtained by evaluating the atoms B and C. (A (B C) C) is a call of A with the first argument obtained from a call of B with argument the value of C, and the second argument the value of C.

To prevent evaluation of arguments the function QUOTE is used. It is a function like any other, except that its argument is not evaluated, since its job is to prevent evaluation of the argument. Thus (A (QUOTE B) (QUOTE C)) is a call of A with arguments B and C, not the values of B and C. Likewise (A (QUOTE (B C)) (QUOTE C)) is a call of A with arguments the list (B C) and the atom C.

S-expressions are extremely inconvenient for complex functions, so Lisp programs are normally written in M-expressions. M is for meta. Function calls in M-expressions look like those of ordinary mathematics. The S-expression (A B C) corresponds to the M-expression a[b;c]. Note the use of brackets and semicolons. The lower case letters are customary in handwritten M-expression, upper case being reserved for S-expressions appearing within the M-expressions. On the teletype, of course, lower case is not available.

2.1 Basic function

Lisp has the following basic functions:

car [x] takes a list as its argument and has the first element as its value. Thus car ['(A B C)'] is A. In M-language the single quote is used in place of the function QUOTE. Car of an atom is an error. Further examples: car ['(((A B) C) (D E F))'] is ((A B) C); car ['((A B C))'] is (A B C).

cdr [x] takes a list as its argument and has the list of the elements after the first as its value. If there is only one element, the value is NIL. cdr of an atom is illegal. Thus cdr ['(A B C)'] is (B C). Note that car ['(A B)'] is A but cdr ['(A B)'] is (B), not B. The value of cdr is always a list unless it is NIL.

cons [x;y] takes two arguments and has the dotted pair of them as its value. Thus cons ['A'; 'B'] is (A.B). Because of the convention for writing lists as dotted pairs, cons ['A';'(B C D)'] is (A B C D). Note that cons is not symmetric in its two arguments: cons ['(B C D)';'A'] is ((B C D).A), not (B C D A). cons[x;y] is also written as x*y.

atom [x] is a predicate. This is, its value is either T or NIL. In particular, atom [x] is T if x is atomic. Thus atom ['A'] is T, atom ['(B C)'] is NIL.

eq [x;y] is another predicate. Its value is T if x and y are the same atom and NIL otherwise. eq[x;y] is also written as x=y.

These are the fundamental operations of Lisp. In addition, there are function definitions and conditionals.

2.2 Function definition

The method of defining functions in M-language will be clear from the following example:

conscar [x;y]: (car x)*y

The list of atoms after the function name is the list of bound variables. When the function is called its actual arguments are evaluated. The old values of the atoms x and y are then saved and their values are set to the values of the arguments. Thus, when they are evaluated in the function definition, the values they provide will be the values of the actual arguments. When the function is finished, the old values of x and y are restored. In this way the function definition is completely insulated from any other uses to which x and y may have been put, and other functions which use x and y need not be concerned about what conscar does to them.

The matter of evaluation and bindings is very important in Lisp and will receive further discussion later. The important points to remember are 1) a function call sets the values of the atoms used in the definition to the values of the actual arguments in the call. The old values are restored afterwards.

2) whenever an expression appears as a function argument it is evaluated. If it is an atom, the evaluation produces its latest binding.

Another way of writing the above function definition makes the binding more explicit:

conscar : lambda [[x ;y]; (car x)*y

Lambda is a function which may be used by the running program to define new functions. It differs from an ordinary function in that it expects its first argument to be a list of atoms and does not evaluate it.

2.3 Conditionals

The one remaining major feature of Lisp is the method for handling conditional branches. This may be illustrated by a function definition:

neq[x;y] :[x=y # NIL; T#T]

The first bracket after the = signals the start of a conditional, which is composed of a number of clauses of the form expression # expression. The first expression in the first clause is evaluated. If its value is not NIL, the second expression is evaluated and its value is the value of the entire conditional. The remaining clauses are ignored. If the first expression turns out to be NIL, the second expression is ignored and attention turns to the second clause. This process proceeds until a first expression is found whose value is not NIL. The absence of such an expression is an error. For this reason it is usual to terminate conditionals with T#, which becomes the catch-all alternative. The reader will observe from this that the value of T is always T. Likewise, the value of NIL is always NIL.

2.4 Recursion

This is not a new feature of Lisp, but it is so important that it deserves special treatment. A function definition may contain calls on any functions, including the one being defined. Because of the machinery for binding variables which has already been discussed, the programmer need not worry about the identity of the arguments. Example:

```
equal[x;y] : [atom  x  #[atom y  #  x=y; T#NIL]; atom y #NIL;
                equal [car x; car y]# equal[cdr x; cdr y]; T#NIL]
```

This function compares two arbitrary lists for equality. The definition says: if x is atomic then if y is atomic the value is eq[x;y], which is already defined. If x is atomic and y not atomic they cannot be equal. If x is not atomic, compare car [x] and car [y]. If they are equal, x and y are equal if cdr [x] and cdr [y] are. If car [x] and car [y] are not equal then x and y are not.

The secret of good Lisp programming is knowing how to use recursion. THINK RECURSIVELY. Recursion can be avoided with PROG, but people who use PROG too much will never get to Heaven.

3.0 Evaluation

What is evaluated? Everything except the argument of QUOTE, the first argument of LAMBDA, SETQ and PROG, and pieces of conditionals which don't get evaluated according to the rules.

What does evaluation do? Atoms are replaced by their most recent binding. Functions have their arguments evaluated and then get called. QUOTE simply disappears and leaves its literal argument. (Note the implications: it is almost always wrong to use QUOTE except on data. If, for instance, you quote a function argument in a definition it will not be replaced by its actual argument, since it will not be evaluated). Lambda and conditionals do peculiar things as described above.

Bindings determine the values of atoms. The value of a number, T or NIL is always what you think it is. All other atoms may have values which are acquired by giving them bindings. The most common way of doing this is to use them as arguments in a definition of a function. When the function is called the atoms used as arguments acquire new bindings and therefore new values. The old values are not lost; they are saved on the pushdown list until the function terminates and then restored. This means that outside the function on the names used for the arguments are not important. Except for one thing: if the function uses eval, which does an extra evaluation of its argument, and the argument of eval when evaluated once turns out to be one of the function arguments, the second evaluation will get the most recent binding, which will not be the one the atom had in the calling function. Thus the definitions

```
dum[x] : x*eval y
redum [y;z] : dum z
```

will cause redum ['(QUOTE A)';'B'] to have the value (B.A), as expected; it will cause redum ['Z';'A'] to have the value (A.A), again as expected; but if re2dum [w;x] : redum [x;w]
then re2dum ['A';'X'] has the value (A.A), not (A.X), because in dum the most recent binding of x is the one produced by the call of dum, namely

A, not the one produced by the call of re2dum, which is X. More realistic examples are also more complex (that is really possible).

There is another way to get bindings, with the function set. set[x;y] evaluates its first argument and expects to get an atom. It then evaluates its second argument and destroys the current binding of eval[x], replacing it with eval[y]. A more useful function is setq, which quotes its first argument. If x is not a function argument in a definition, then setq[x;'A'] sets the value of x to A permanently. If x is a function argument, then the setq overrides the current binding, but when that binding is destroyed by the termination of the function whose call produced it, the effect of the setq is also lost.

When the user types in to Lisp at the console, he is talking to eval. More precisely, what he types is substituted for X in the form (PRINT X), where PRINT is a function which prints its argument. This means that eval is applied once to whatever is typed. Hence, typing an atom causes its value to be printed out. If the atom is a function name, its value is the S-expression function definition.

Typing (CONS A B) probably gives an error, unless A and B have been setq'ed. On the other hand (CONS (QUOTE A) (QUOTE B)) gives (A.B).

4.0 Arithmetic

There are a few Lisp functions for operations on numbers. Plus takes any number of arguments and adds them together to get its result. Times multiplies its arguments. Quot divides the first argument by the second. Rem takes the first argument modulo the second. Note that because arguments are evaluated before the functions are called, the expression (a+b) (c+d) is evaluated in Lisp by times [plus[a;b];plus[c;d]].

To compare numbers the predicate gtp is available; it is true if the first argument is greater than the second. The predicate numbp is true if its argument is a number. All of these functions except numbp give errors if their arguments are non-numeric.

Minus is a function of one numeric argument which returns the negative of the argument as its value.

5.0 Standard functions

The following functions are either built into the 930 Lisp system or in the library:

list takes its arguments and strings them together into a list. It is one of the few standard functions which take a variable number of arguments.

member [x;y] is true if x is a member of the list y. The definition is

```
member [x;y] : [null y #NIL; equal [x;car y ]#T; T# member [x;cdr y]]
```

equal [x;y] is true if x equals y. It was defined above.

subst [x;y;z] substitutes x for all occurrences of the S-expression y in z.

```
subst [x;y;z] : [atom z #[ z=y #X;T#z]; T# subst
      [x;y;car z]* subst [x;y;cdr z]]
```

sassoc [x;a;u] searches the pair list a for a dotted pair whose first element is x and returns this pair as its value. A pair list is a list of dotted pairs. If there is no such pair the value is the value of the function u of no arguments.

```
sassoc [x;a;u] : [null a # u[ ]; (caar a)=x #car a; T#sassoc[x;cdr a ;u]]
```

pair [x;y] has as value the list of pairs of corresponding elements of the lists x and y. It can be used to construct pair lists of the kind searched by sassoc.

```
pair[x;y]: [null x#NIL; T#((car x)*car y)*pair[cdr x;cdr y]]
```

append [x;y] combines its two arguments into one new list. It is not like cons.

```
append [x;y] : [null x #y;T#(car x)*append[cdr x;y]]
```

and takes any number of arguments and is T if none of them is NIL, NIL otherwise.

or takes any number of arguments and is NIL if all of them are NIL, T otherwise.

null [x] is T if x is NIL or the empty list (), which is the same thing. Otherwise it is NIL. null is exactly the same as not, which is therefore not provided.

gensym[ ] has as value a symbol guaranteed different from any other in the world.

length [x] has the number of elements in x as its value.

reverse [x] has as value the list with the elements of x in reverse order.

eval has already been discussed. It evaluates its argument again. Do not forget that the argument is always evaluated once.

prog2 [x;y] has as its value the value of y. Its function is to get both x and y evaluated.
caar [x] : car car x
cadr [x] : car cdr x
cdar [x] : cdr car x
cddr [x] : cdr cdr x
caaar [x] through cdddr [x] are defined similarly.

define [x] takes as its single argument a list of things to be defined. Each of these things is either a list of two elements:

(function name S-expression)

which sets the literal S-expression to be the value of the function name; or a list of three elements:

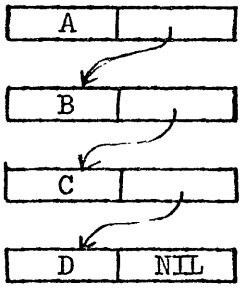
(function name (list of variables) S-expression)

which sets the value of the function name to (LAMBDA (list of variables) S-expression).
Each element of the first kind is equivalent to
function name : 'S-expression'
and each element of the second kind is equivalent to
function name [list of variables] : S-expression.

function [x] is exactly equivalent to quote [x].

6.0 Changing list structure

A dotted pair in memory is a word with two pointers, one to the first element of the pair and the other to the second. The list (A B C D) then looks like this:



Because of this use of pointers, a list may be a member of many other lists, each of which has a pointer to it. When a new list is generated by cons, a single memory word is used. In it are placed pointers to the two elements of the dotted pair being created by the cons. No account is taken of other lists of which either element may be a member.

These considerations are important for proper use of two functions which explicitly change pointers in an existing list structure, since these functions can affect all the lists which have pointers to the cell being changed. If the programmer is not alert, he may not be aware of how many such lists there are.

rplaca [x;y] replaces the first element of the dotted pair x with a pointer to y.

rplacd [x;y] replaces the second element of the dotted pair x with a pointer to y.

rplaca looks something like y*cdr x, but it is not the same. It does not create a new word, and it permanently changes x.

A useful function which makes use of these operations is nconc, which is like append except that it does not copy its first argument. For this reason it is somewhat more efficient than append. It is also much more dangerous. Unless you are very sure of what you are doing, do not use

nconc except when the first argument has just been created by list or some similar function. Nconc does a rplacd on the last element of its first argument.

7.0 Functionals

It is possible, as the discerning reader will already have noticed, for a function to have functions as arguments. Such functions are called functionals, and they are very useful. The most useful ones are the true Lisp programmer's substitute for iteration; they are the mapping functionals.

maplist [x;f] runs down the list x and applies the function f to each sublist obtained by taking elements off the front of x. The values are then cons'ed together. The definition is clearer: maplist [x;f] : [null x #NIL;T#
(f x)* maplist[cdr x ;f]]

Since this is rarely exactly what is wanted, there are several other mapping functions which may be more suitable for particular applications.
mapcon [x;f] : [null x #NIL;T#nconc[f x ; mapcon[cdr x ;f]]]
is like maplist except that it uses nconc rather than cons. It is not safe if the value of f is a list the last element of which is on any other lists.
map[x;f] : [null x #NIL;T#prog2[f x ; map[cdr x ;f]]] is like maplist except that it does not save the values of f. It is good when f is being executed for its effect rather than its value.
mapcar [x;f] : [null x #NIL; T#(f car x)* mapcar[cdr x ;f]] is like maplist except that it applies f to each element of x in turn, instead of to each tail in turn.

8.0 Prog

As we have already mentioned, there is in Lisp a feature which allows the programmer to write sequences of statements, just like Fortran, and transfer between them. To do this, use the pseudo-function prog, thus

```
reverse [x] : prog [[y] ;
a;          [null x # return y ];
            y ←(car x)*y;
            x ← cdr x;
            go a ]
```

which defines the library function reverse. Note that statements are separated by semicolons. The program variables are specified in the first list of the prog definition; they are bound by the prog and are set to NIL when it is entered. They may be regarded as function arguments which are always NIL when the function is called.

Labels of statements are atoms followed by semicolons. Transfers are done with the function go. To leave the prog, execute the function return , which delivers its argument as the value of the prog. If the prog is left because the last statement was executed and did not include a go, its value is NIL.

The function setq takes the place of the Fortran assignment statement. It is of course the same function that is available outside of progs to set variables. This conclusion follows from the general rule that a statement in a prog may be any S- or M-expression which would be legal as the argument of a function. The one exception is that a conditional used alone as a statement is permitted to run off the end. Progs may of course be nested and may in fact appear wherever any other function call is legal.

9.0 Property Lists

An atom in Lisp is like a hatrack with hooks on which various things can be hung. Some of these have already been discussed: the value, which is an S-expression; and the print name, which is the name of the atom and not accessible to the programmer except on input-output.

There is one more hook which is not used by the Lisp system itself for anything. This is the property list. It is an S-expression like the value, and its only function is to provide a convenient place to keep information about the atom. There are two functions connected with it.

setlis [x;y] sets the property list of the atom x to be the S-expression y.

getlis [x] has as its value the property list of the atom x.

10.0 Input-output

Lisp does input-output with a small number of useful functions. read is a function of no arguments which reads a single S-expression from the current input medium. Read will treat any punctuation character except ( ) " and dot as an atom. It will also take any string of characters enclosed in a double quote as an atom. The first character of the string is not checked for double quote, i.e., to input " as atom write """".

print [x] prints the S-expression x on the current output medium.

prinl [x] prints the single atom x on the current output medium.

input [x] sets the input medium to the file whose name is the atom x.

output [x] does the same for the output medium.

The input medium can also be set from the teletype. Both input and output media are reset to teletype by pushing the rubout button and by any error. When the input or output medium is switched, the former input or output file is closed (unless it is the teletype). File names need not be quoted.

terpri is a function of no arguments which prints a carriage return and line feed.

11.0 Other useful functions

trace [x] takes a list of function names and changes the definitions so that the function name and the arguments are printed each time the function is called, and the function name and the value are printed each time the function exits.

untrace [x] reverses the action of trace. Untracing functions which have not been traced is likely to cause an error. If it does not, the function definition will not be correct afterwards.

nlamda [[x]; definition] is like lambda with the following exception: When the function defined by the nlamda is called, its arguments are collected literally and made into a list. x is then bound to this list. The arguments are not evaluated. Using nlamda it is possible to write so-called pseudo-functions like quote, which do not follow the usual Lisp rules. In fact, quote is defined by

```
quote [x] : nlamda [[x]; car  x ]
```

12.0 Error comments

Errors detected by the interpreter cause three character error comments and return control to the Lisp supervisor. The error comments are:

Code	additional information	meaning
IAR	atom	tried to take CAR or CDR of an atom
ICD		ran off the end of a COND
ILS	first arg	tried to SETQ a non-atom or number T or NIL
IRP	first arg	tried to RPLACA or RPLACD an atom
ISG	first arg	tried to SETLIS or GETLIS a non-atom
NNA	argument	non-numeric argument for an arithmetic function
PCE		ran off pushdown list
SCE		storage capacity exceeded
???		disaster. give up
IAF	function	illegal atom used as function name
INA		wrong number of arguments for built-in function
INL		wrong number of arguments for LAMBDA
UAS	atom	unbound atom evaluated
GCH		garbage collector snarled. give up
IIF	line	) or . at the beginning of an S-expression
IIP	line	. not followed by (or atom
IIT	line	. atom not followed by)
INM	line	illegal number
PR1	argument	PRIN1 called with non-atomic argument

13.0 Special features of the M-language translator

The internal operation of Lisp is exclusively in S-expressions, but in practice all input and output of programs is done in M-language. Each list read from the input may contain either a single function definition or a single function call to be evaluated. Thus, to define cadr the following teletype input might be used:

```
$( CADR  X  : CAR  CDR X)
```

to which Lisp would respond

```
( CADR )
$
```

The translator types $ to indicate that it is waiting for more input.

The M-language translator has a number of useful features, some of which are not yet implemented. The form which is now implemented is the one used in the function definitions in this manual; that is, it has a limited number of binary operators (: for function definition, = for eq, ← for setq, * for cons) and accepts no special symbols except single quotes. There are three exceptions:

1) Double quotes may be used to input unmentionable characters as described in section 9.
2) A function of one argument may be written without brackets surrounding the argument. Thus car cdr car x is equivalent to car [cdr[car[x]]].
3) Parentheses have their usual mathematical meaning.

A useful function for examining function definitions is see [x] where x is either a single function name or a list of names. This function prints out the definitions in M-language.

Binary operators should be used with caution: they take the shortest possible left operand and the largest possible right operand:

car x*car y	is	car[cons[x;car[y]]]
car[x]*car y	is	cons[car[x];car[y]]
car(x*y)*z	is	car[cons[cons[x;y];z]]

14.0 The Lisp Operating System

To call in the Lisp system give the executive command

@ LISP.

The Lisp system will then read in the library and type $. At this point the M-language translator is in control and remains in control until either control returns to the exec, or the user types $ himself. In either case Lisp will be in S-language mode: each S-expression read in will be evaluated and the result printed out, as discussed in section 3. To return to M-language, type (XM).

To change the input medium, type bell (control G), followed by a file name. Data is read from this file until an end of file is encountered, after which control returns to the teletype. Pushing the rubout button always returns control to the teletype.

During input of text, A^c (control A) deletes the most recently typed character in the current line. If there are no characters left in the current line, it has no effect. Q^c deletes the current line completely. Once a carriage return has been typed, nothing can delete the line except the rubout button. Carriage return looks the same as space, and the system always provides a line feed.

Whenever the Lisp system is doing anything, pushing the rubout button terminates the activity and returns control to the teletype. If this happens during computation, some atoms may have rather strange bindings. Other than this, no trouble can be caused by pushing the rubout button.

After the rubout button has been pushed, pushing it again without typing anything else will cause control to revert to the exec. To return to Lisp without initializing the system, give the exec command

@ CONTINUE LISP.

Appendix 1: Functions is the Lisp system

The following functions are machine coded in the Lisp system.

The following functions are in the Lisp library