LISP data are symbolic expressions that can be either atoms or lists. Atoms are strings of letters and digits and other characters not otherwise used in LISP. A list consists of a left parenthesis followed by zero or more atoms or lists separated by spaces and ending with a right parenthesis. Examples: A, ONION, (), (A ONION A), (PLUS 3 (TIMES X PI) I), (CAR (QUOTE (A B))).

The LISP programming language is defined by rules whereby certain LISP expressions have other LISP expressions as values. The function called value that we will use in giving these rules is not part of the LISP language but rather part of the informal mathematical language used to define LISP. Likewise, the italic letters $v$ and $a$ (sometimes with subscripts) denote LISP expressions, the letter $v$ (usually subscripted) denotes an atom serving as a variable, and the letter $f$ stands for a LISP expression serving as a function name.

1. value (QUOTE $e$) = $e$. Thus the value of (QUOTE A) is A.

2. value (CAR $e$), where value $e$ is a non-empty list, is the first element of value $e$. Thus value (CAR (QUOTE (A B C))) = A.

3. value (CDR $e$), where value $e$ is a non-empty list, is the the list that remains when the first element of value $e$ is deleted. Thus value (CDR (QUOTE (A B C))) = (B C).

4. value (CONS $e_1$ $e_2$), is the list that results from prefixing value $e_1$ onto the list value $e_2$. Thus value (CONS (QUOTE A) (QUOTE (B C))) = (A B C).

5. value (EQUAL $e_1$ $e_2$) is T if value $e_1$ = value $e_2$. Otherwise, its value is NIL. Thus value (EQUAL (QUOTE (A B)) (QUOTE A)) = T.

6. value (ATOM $e$) = T if value $e$ is an atom; otherwise its value is NIL.

7. value (COND($p_1$ $e_1$) ... ($p_n$ $e_n$)) = value $e_i$, where $p_i$ is the the first of the $p$'s whose value is not NIL. Thus value (COND ((ATOM (QUOTE A)) (QUOTE B)) (QUOTE T)) (QUOTE C)) = B.

8. An atom $v$, regarded as a variable, may have a value.

9. value ((LAMBDA ($v_1$ ... $v_n$) $e$) $e_1$ ... $e_n$) is the same as value ((LAMBDA ($v_1$ ... $v_n$) $e$) $e_1$ ... $e_n$) with the additional rule that whenever ($f$ $a_1$ ... $a_n$) must be evaluated, $f$ is replaced by (LABEL $f$ (LAMBDA ($v_1$ ... $v_n$) $e$)). Lists beginning with LABEL define functions recursively.

10. Here's the hard one. value ((LABEL $f$ (LAMBDA ($v_1$ ... $v_n$) $e$)) $e_1$ ... $e_n$) is the same as value ((LAMBDA ($v_1$ ... $v_n$) $e$) $e_1$ ... $e_n$) with the additional rule that whenever ($f$ $a_1$ ... $a_n$) must be evaluated, $f$ is replaced by (LABEL $f$ (LAMBDA ($v_1$ ... $v_n$) $e$)). Lists beginning with LABEL define functions recursively.

This is the core of LISP, and here are more examples:

value (CAR X) = (A B) if value X = ((A B) C), and value ((LABEL FF (LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR X)))))) (QUOTE ((A B) C))) = A. Thus ((LABEL FF (LAMBDA (X) (COND ((ATOM X) X) ((QUOTE T) (FF (CAR X)))))) is the LISP name of a function ff such that ff $e$ is the first atom in the written form of $e$. Note that the list ff is substituted for the atom FF twice.

Difficult mathematical type exercise: Find a list $e$ such that value $e$ = $e$.

Abbreviations

The above LISP needs some abbreviations for practical use.

1. The variables T and NIL are permanently assigned the values T and NIL, and NIL is the name of the null list ()

2. So as not to describe a LISP function each time it is used, we define it permanently by typing (DEFUN f $e$ $e$). Thereafter $(f$ $e_1$ ... $e_n$) is evaluated by evaluating $e$ with the variables $v_1$, ..., $v_n$ taking the values value $e_1$, ..., value $e_n$, respectively. Thus, after we define (DEFUN FF (X) (COND ((ATOM X) X) (T (FF (CAR X))))), typing (FF (QUOTE (CA B) C))), gets A from LISP.

3. We have the permanent function definitions

(DEFUN NULL (X) (EQUAL X NIL)) and

(DEFUN CADR (X) (CAR (CDR X))),

and similarly for arbitrary combinations of A and D.

4. (LIST $e_1$ ... $e_n$) is defined for each $n$ to be (CONS $e_1$ (CONS ... (CONS $e_n$ NIL))).

5. (AND $p$ $q$) abbreviates (COND ($p$ $q$) (T NIL)). ANDs with more terms are defined similarly, and the propositional connectives OR and NOT are used in abbreviating corresponding conditional expressions.

Here are more examples of LISP function definitions:
(DEFUN ALT (X) (COND ((OR (NULL X) (NULL (CDR X)))
X) (T (CONS (CAR X) (ALT (CDDR X))))))

defines a function that gives alternate elements of a list starting
with the first element. Thus (ALT (QUOTE (A B C D E))) = (A C E).

(DEFUN SUBST (X Y Z) (COND ((ATOM Z) (COND ((EQUAL
Z Y) X) (T Z))) (T (CONS (SUBST X Y (CAR Z)) (SUBST X Y
(CDR Z)))))).

where Y is an atom, gives the result of substituting X for Y in Z. Thus

(SUBST (QUOTE (PLUS X Y)) (QUOTE V) (QUOTE (TIMES
X Y))) = (TIMES X (PLUS X Y)).

You may now program in LISP. Call LISP on a time-
sharing computer, define some functions, type in a LISP
expression, and LISP will output its value on your terminal.

THE LISP INTERPRETER WRITTEN IN LISP

The rules we have given for evaluating Lisp expressions

can themselves be expressed as a LISP function (EVAL e a), where e is an expression to be evaluated, and a is a list of
variable-value pairs. a is used in the recursion and is often initially NIL. The long LISP expression that follows is just such an
evaluator. It is presented as a single LABEL expressions with
all auxiliary functions also defined by LABEL expressions
internally, so that it references only the basic function of LISP and
some of abbreviations like CADR and friends. It knows about all
the functions that are used in its own definition so that it can
evaluate itself evaluating some other expression. It does not know
about DEFUNs or any features of LISP not explained in this
micro-manual such as functional arguments, property list functions,
input-output, or sequential programs.

The function EVAL can serve as an interpreter for LISP, and LISP interpreters are actually made by hand-compiling EVAL
into machine language or by cross-compiling it on a machine for
which a LISP system already exists.

The definition would have been easier to follow had we
defined auxiliary functions separately rather than include them
using LABEL. However, we would then have needed property list
functions in order to make the EVAL self-applicable. These
auxiliary functions are EVLIS which evaluates lists of expressions,
EVCOND which evaluates conditional expressions, ASSOC which
finds the value associated with a variable in the environment, and
PAIRUP which pairs up the corresponding elements of two lists.

Here is EVAL.

(DEFUN ALT (X) (COND ((OR (NULL X) (NULL (CDR X)))
X) (T (CONS (CAR X) (ALT (CDDR X))))))

defines a function that gives alternate elements of a list starting
with the first element. Thus (ALT (QUOTE (A B C D E))) = (A C E).

(DEFUN SUBST (X Y Z) (COND ((ATOM Z) (COND ((EQUAL
Z Y) X) (T Z))) (T (CONS (SUBST X Y (CAR Z)) (SUBST X Y
(CDR Z)))))).

where Y is an atom, gives the result of substituting X for Y in Z. Thus

(SUBST (QUOTE (PLUS X Y)) (QUOTE V) (QUOTE (TIMES
X Y))) = (TIMES X (PLUS X Y)).

You may now program in LISP. Call LISP on a time-
sharing computer, define some functions, type in a LISP
expression, and LISP will output its value on your terminal.

THE LISP INTERPRETER WRITTEN IN LISP

The rules we have given for evaluating Lisp expressions

can themselves be expressed as a LISP function (EVAL e a), where e is an expression to be evaluated, and a is a list of
variable-value pairs. a is used in the recursion and is often initially NIL. The long LISP expression that follows is just such an
evaluator. It is presented as a single LABEL expressions with
all auxiliary functions also defined by LABEL expressions
internally, so that it references only the basic function of LISP and
some of abbreviations like CADR and friends. It knows about all
the functions that are used in its own definition so that it can
evaluate itself evaluating some other expression. It does not know
about DEFUNs or any features of LISP not explained in this
micro-manual such as functional arguments, property list functions,
input-output, or sequential programs.

The function EVAL can serve as an interpreter for LISP, and LISP interpreters are actually made by hand-compiling EVAL
into machine language or by cross-compiling it on a machine for
which a LISP system already exists.

The definition would have been easier to follow had we
defined auxiliary functions separately rather than include them
using LABEL. However, we would then have needed property list
functions in order to make the EVAL self-applicable. These
auxiliary functions are EVLIS which evaluates lists of expressions,
EVCOND which evaluates conditional expressions, ASSOC which
finds the value associated with a variable in the environment, and
PAIRUP which pairs up the corresponding elements of two lists.