Lisp 1.5 Programmer's Manual

The LISP Language

- The LISP language is designed primarily for symbolic data processing.
- LISP is a formal mathematical language.
- LISP differs from most programming languages in three important ways. The first way is in the nature of the data. In the LISP language, all data are in the form of symbolic expressions usually referred to as S-expressions. S-expressions are of indefinite length and have a branching tree type of structure. In the LISP programming system, the bulk of available memory is used for storing S-expressions in the form of list structures. This type of memory organization frees the programmer from the necessity of allocating storage for the different sections of his program.

  The second important part of the LISP language is the source language itself which specifies in what way the S-expressions are to be processed. This consists of recursive functions of S-expressions. Since the notation for the writing of recursive functions of S-expressions is itself outside the S-expression notation, it will be called the meta language. These expressions will therefore be called M-expressions.

  Third, LISP can interpret and execute programs written in the form of S-expressions. Thus, like machine language, and unlike most other higher level languages, it can be used to generate programs for further execution.

S-expressions (Symbolic expressions)

- The most elementary type of S-expression is the atomic symbol. An atomic symbol is a string of no more than thirty numerals and capital letters; the first character must be a letter.
- All S-expressions are built out of atomic symbols and the punctuation marks `.`, `(` and `)`. The basic operation for forming S-expressions is to combine two of them to produce a larger one. An S-expression is either an atomic symbol or it is composed of these elements in the following order: a left parenthesis, an S-expression, a dot, an S-expression, and a right parenthesis. From the two atomic symbols `A1` and `A2`, one can form the S-expression `(A1.A2)`.
- LISP has an alternative form of S-expression called the list notation. The atomic symbol `NIL` serves as a terminator for lists. The null list `( )` is identical to `NIL`. Lists may have sublists. The dot notation and the list notation may be used in the same S-expression. Historically, the separator for elements of lists was the comma; however, the blank is now generally used. The two are entirely equivalent in LISP. `(A, B, C)` is identical to `(A B C)`. Examples:

```
(A) = (A . NIL)
((A)) = ((A . NIL) . NIL)
(A B C) = (A . (B . (C . NIL)))
(A (B . C)) = (A . ((B . C) . NIL))
((A B) C) = ((A . (B . NIL)) . (C . NIL))
(A B (C D)) = (A . (B . ((C . (D . NIL)) . NIL)))
```

Functions

- We shall introduce some elementary functions of S-expressions. To distinguish the functions from the S-expressions themselves, we shall write function names in lower case letters, since atomic symbols consist of only upper case letters. Furthermore, the arguments of functions will be grouped in square brackets rather than parentheses. As a separator or punctuation mark we shall use the semicolon.

- Function `cons`:

  `cons[A;B] = (A.B)`
  `cons[(A.B);C] = ((A.B).C)`
  `cons[cons[A;B];C] = ((A.B).C)`

  It is possible to build any S-expression from its atomic components by compositions of the function `cons`.

- Function `car`:

  `car[A] is undefined`
  `car[(A.B)] = A`
  `car[(A.(B1.B2))] = A`
  `car[((Al.A2).B)] = (A1.A2)`

  The `car` function does just the opposite of `cons`. It produces the subexpressions of a given expression.

- Function `cdr`:

  `cdr[A] is undefined`
  `cdr[(A.B)] = B`
  `cdr[(A.(B1.B2))] = (B1.B2)`
  `cdr[((A1.A2).B)] = B`

  The `cdr` function is similar to `car`. Its value is the second part of its composite argument.

- We can produce S-expressions by a suitable composition of `cons`, `car` and `cdr`. For instance, an identity function for `x` is `car[cons[x;y]] = x`. The following identity is also true for any S-expression `x` such that `x` is composite (non- atomic): `cons[car[x];cdr[x]] = x`.

- In LISP, variables are used to represent S-expressions. In choosing names for variables and functions, we shall use the same type of character strings that are used in forming atomic symbols, except that we shall use lower case letters.

- A function whose value is either true or false is called a predicate. In LISP, the values true and false are represented by the atomic symbols `T` and `F`, respectively. A LISP predicate is therefore a function whose value is either `T` or `F`.

- Predicate `eq`:

  `eq[A;A] = T`
  `eq[A;B] = F`
  `eq[A;(A.B)] is undefined`
  `eq[(A.B);(A.B)] is undefined`

  Predicate `eq` is a test for equality on atomic symbols. It is undefined for non-atomic arguments.

- Predicate `atom`:

  `atom[EXTRALONGSTRINGOFLETTERS] = T`
  `atom[(U.V)] = F`
  `atom[car[(U.V)]] = T`

  The predicate `atom` is true if its argument is an atomic symbol and false if it is composite.

- It is important to become familiar with the results of elementary functions on S-expressions written in list notation. These can always be determined by translating into dot notation. Examples:

  `cdr[(A)] = cdr[(A.NIL)] = NIL`
  `car[(A B C)] = car[(A . (B C))] = A`
  `cdr[(A B C)] = cdr[(A . (B C))] = (B C)`
  `car[cdr[(A B C)]] = B`
  `car[((A B) C)] = (A B)`
  `cons[A; (B C)] = (A B C)`

- It is convenient to abbreviate multiple `car`'s and `cdr`'s. This is done by forming function names that begin with `c`, end with `r`, and have several `a`'s and `d`'s between them. Examples:

  `cadr[(A B C)] = car[cdr(A B C)] = B`
  `caddr[(A B C)] = C`
  `cadadr[(A (B C) D)] = C`

The last a or d in the name actually signifies the first operation in order to be performed.

The LISP Meta-language

- Functions are part of the LISP Meta-language, they are M-expressions.
- Function names and variable names are like atomic symbols except that they use lower case letters.
- The arguments of a function are bound by square brackets and separated from each other by semicolons.
- Compositions of functions may be written by using nested sets of brackets. These rules allow one to write function definitions such as: `third[x] = car[cdr[cdr[x]]] = caddr[x]`.
- The class of functions that can be formed in this way is quite limited and not very interesting. A much larger class of functions can be defined by means of the conditional expression, a device for providing branches in function definitions. A conditional expression has the following form:

  `[p1 -> e1; p2 -> e2; ... ; pk -> ek]`

  This means that if `p1` is true, the value `e1` is the value of the entire expression. If `p1` is false, then if `p2` is true, the value `e2` is the value of the entire expression. And so on. If none of the `pi` are true the value of the entire expression is undefined. Example:

  `[eq[car[x];A] -> cons[B;cdr[x]]; T -> x]`

  The main application of conditional expressions is in defining functions recursively. Example:

  `ff[x] = [atom[X] -> X; T -> ff[car[X]]]`

  The definition of `ff` is recursive in that `ff` is actually defined in terms of itself.
- A recursive function can be infinitely recursive for certain arguments. When such a function is interpreted in the LISP programming system, it will either use up all of the available memory, or loop until the program is halted artificially.
- Other functions using conditional expressions:

  Absolute value: `|x| = [ x < 0 -> -x; T -> x]`

  Factorial: `n! = [n=0 -> 1; T -> n * [n – 1]! ]`

  A detailed discussion of the theory of functions defined recursively by conditional expressions is found in "A Basis for a Mathematical Theory of Computation" by J. McCarthy in 1961.

Relation with Alonzo Church's Lambda

- It is usual for most mathematicians to use the word function imprecisely, and to apply it to forms such as `y*y+x`. We need a notation that expresses the distinction between functions and forms: the lambda notation of Alonzo Church.
- Let `f` be an expression that stands for a function of two integer variables. It should make sense to write `f[3;4]` and to be able to determine the value of this expression. For example, `sum[3;4]` is `7`. The expression `y*y+x` does not meet this requirement. It is not clear whether the value is `13` or `19`. The expression `y*y+x` is a form but not a function, because we cannot determine its value. A form can be converted to a function by specifying the correspondence between the variables in the form and the arguments of the desired function.
- If `δ` is a form in the variables `x1; ...; xn` then the expression `λ[[x1;...;xn];δ]` represents the function of n variables obtained by substituting the n arguments in order for the variables `x1`, ..., `xn`. For instance, the function `λ[[x;y];y*y+x]` is a function of two variables for the form `y*y+x`, and we substitute the arguments into values as in: `λ[[x;y];y*y+x][3;4] = 4*4+3 = 19`.

- The variables in a lambda expression are dummy or bound variables because systematically changing them does not alter the meaning of the expression. Thus, `λ[[u;v];v*v+u]` means the same thing as `λ[[x;y];y*y+u]`.
- We shall use expressions in which a variable is not bound by a lambda. For instance, the variable `n` in the function `λ[[x;y];n*(x+y)]`. Here, `n` is called a free variable. Unless `n` has been given a value before trying to compute with this function, the value of the function must be undefined.
- For naming recursive functions, not only must the variables be bound, but the name of the function must be bound. Using lambda notation we can define

  `ff[x] = [atom[x] -> x; T -> ff[car[x]]]`

  As follows:

  `ff = λ[[x]; [atom[x] -> x; T -> ff[car[x]]]]`

  However, this notation is incosistent with LISP's syntactic rules. We need something to indicate that, in the lambda expression of the previous equation, the occurrence of `ff` stands for the function that is being defined. For this endeavor, we introduce the `label` notation. If `δ` is an expression, and `α` is its name, we write `label[α;δ]`. Now we can define the function `ff` as follows:

  `label[ff; λ[[x]; [atom[x] -> x; T -> ff[car[x]]]]`

  In this expression, `x` is a bound variable and `ff` is a bound function name.
- All parts of the LIPS language have now been explained.

Evaluation of LISP expressions

- An interpreter or universal function is one that can compute the value of any given function applied to its arguments when given a description of that function. If the function being interpreted has infinite recursion, the interpreter will recur infinitely too.
- We define the interpreter function as the universal LISP function `evalquote[fn;args]`. When `evalquote` is given a function `fn` and a list of arguments `args` for that function, it computes the value of the function `fn` applied to the given `args` arguments.

  Because every LISP function has S-expressions as arguments, in particular, the function `fn` must also be an S-expression (as well as every argument in `args`). However, so far we have been writing functions like `fn` as M-expressions. It is necessary to translate them into S-expressions.
- We use the following rules for translating functions (M-expressions) into S-expressions:

  1. Function names are translated by changing all of the letters to upper case, making it an atomic symbol. For instance, `car` (M-expression) is translated to `CAR` (S-expression).

  2. A variable is also translated by using uppercase letters. Thus, the translation of `car[x]` is `(CAR X)`.

  3. We quote constants to translate them, to distinguish it from variables. Thus, the truth constant `T` is translated into `(QUOTE T)`.

  4. The form `fn[arg1;...;argn]` is translated into `(fn arg1 ... argn)`, where `fn` is the translation of M-expression `fn` into S-expression `fn`; `arg1` is the translation of M-expression `arg1`; and so on.

  5. The conditional expression `[p1 -> e1; ...; pn -> en]` is translated into the following S-expression `(COND (p1 e1) ... (pn en))`, where `p1` is the translation of M-expression `p1`, `e1` is the translation of M-expression `e1`, and so on.

  6. Lambda notation `λ[[x1;...;xn];δ]` is translated into `(LAMBDA (X1 ... XN) δ)`, where `δ` is the translation of M-expression `δ` into an S-expression `δ`.

  7. Label notation `label[α;δ]` is translated into `(LABEL α δ)`, where `α` is the translation of M-expression `α` into an S-expression `α`, and `δ` into `δ`.

- We now define `evalquote[fn;args]` as follows. Let `fn` (S-expression) be the translation of `fn` (M-expression). Let `(arg1 ... argn)` be a list of n S-expressions, and let `args=(arg1 ... argn)`. Then

  `evalquote[fn;args] = fn[arg1;...;argn] = (fn arg1 ... argN)`

  Only if either side of the equation is defined at all. Example:

```
evalquote[λ[[x;y];cons[car[x];y];((A B)(C D))] =
evalquote[(LAMBDA (X Y) (CONS (CAR x) Y));((A B) (C D))] =
λ[[x;y];cons[car[x];y]][(A B);(C D)] =
cons[car[(A B)];(C D)] =
(A C D)
```

- `evalquote` is defined by using two main functions, called `eval` and `apply`.
- `apply` handles a function and its arguments (it applies a function), while `eval` handles forms (it evaluates an expression). Each of these functions also has another argument that is used as an association list for storing the values of bound variables and function names.

```
evalquote[fn; x] = apply[fn; x; NIL]
```

- Here is the implementation of `apply`.

```
apply [fn; x; a] =
[atom[fn] -> [eq[fn; CAR] -> caar[x];
              eq[fn; CDR] -> cdar[x];
              eq[fn; CONS] -> cons[car[x]; cadr[x]];
              eq[fn; ATOM] -> atom[car[x]];
              eq[fn; EQ] -> eq[car[x]; cadr[x]];
              T -> apply[eval[fn; a]; x; a]];
eq[car[fn]; LAMBDA] -> eval[caddr[fn]; pairlis[cadr[fn]; x; a]];
eq[car[fn]; LABEL] -> apply[caddr[fn]; x; cons[cons[cadr[fn]; caddr[fn]]; a]]]
```

- Here is the implementation of `eval`.

```
eval[e; a] = [atom[e] -> cdr[assoc[e; a]];
              atom[car[e]] ->
                   [eq[car[e]; QUOTE] -> cadr[e];
                   eq[car[e]; COND] -> evcond[cdr[e]; a];
                   T -> apply[car[e]; evlist[cdr[e]; a]; a]];
              T -> apply[car[e]; evlist[cdr[e]; a]; a]]
```

- Here are the auxiliary function definitions:

```
pairlis[x; y; a] = [null[x] -> a;
  T -> cons[cons[car[x]; car[y]]; pairlis[cdr[x]; cdr[y]; a]]]

assoc[x; a] = [equal[caar[x]; x] -> car[a];
               T -> assoc[x; cdr[a]]]

evcond[c; a] = [eval[caar[c]; a] -> eval[cadar[c]; a];
                T -> evcond[cdr[c]; a]]

evlist[m; a] = [null[m] -> NIL;
                T -> cons[eval[car[m]; a]; evlist[cdr[m]; a]]]
```

- The first argument for `apply` is a function. If it is an atomic symbol, then there are two possibilities. One is that it is an elementary function: `car`, `cdr`, `cons`, `eq`, or `atom`. In each case, the appropriate function is applied to the argument(s). If it is not one of these, then its meaning has to be looked up in the association list.
- If it begins with `LAMBDA`, then the arguments are paired with the bound variables, and the form is given to `eval` to evaluate.
- If it begins with `LABEL`, then the function name and definition are added to the association list, and the inside function is evaluated by `apply`.
- The first argument of `eval` is a form. If it is atomic, then it must be a variable, and its value is looked up on the association list.
- If `car` of the form is `QUOTE`, then it is a constant, and the value is `cadr` of the form itself.
- If `car` of the form is `COND`, then it is a conditional expression, and `evcond` evaluates the propositional terms in order, and chooses the form following the first true predicate.
- In all other cases, the form must be a function followed by its arguments. The arguments are then evaluated, and the function is given to `apply`.
- This concludes the presentation of the purely formal mathematical system that we shall call pure LISP. The elements of this formal system are:

  1. A set of symbols called S-expressions.

  2. A functional notation called M-expressions.

  3. A formal mapping of M-expressions into S-expressions.

  4. A universal function (written as an M-expression) for interpreting the application of any function written as an S-expression to its arguments.

The LISP Interpreter System

- The basis of the LISP Programming System is the interpreter, or `evalquote` and its components. A LISP program in fact consists of pairs of arguments for `evalquote` which are interpreted in sequence.
- A program for execution in LISP consists of a sequence of doublets. The first list or atomic symbol of each doublet is interpreted as a function. The second is a list of arguments for the function. They are evaluated by `evalquote`, and the value is printed. For instance,

  ```
  (LAMBDA (X Y) (CONS X Y)) (A B)
  ```

- Do not use the forms `(QUOTE T)`, `(QUOTE F)`, and `(QUOTE NIL)`. Use `T`, `F`, and `NIL` instead.
- Atomic symbols should begin with alphabetical characters to distinguish them from numbers.
- A selection of basic functions is provided with the LISP system. Other functions may be introduced by the programmer. The order in which functions are introduced is not significant. Any function may make use of any other function.
- The formalism for variables in LISP is the Church lambda notation. The part of the interpreter that binds variables is called `apply`. When `apply` encounters a function beginning with `LAMBDA`, the list of variables is paired with the list of arguments and added to the front of the `a`-list . During the evaluation of the function, variables may be encountered. They are evaluated by looking them up on the `a`-list. If a variable has been bound several times, the last or most recent value is used. The part of the interpreter that does this is called `eval`.
- A pseudo-function is a function that is executed for its effect on the system in core memory, as well as for its value.

- In LISP, a variable remains bound within the scope of the LAMBDA that binds it. When a variable always has a certain value regardless of the current a-list, it will be called a constant. This is accomplished by means of the property list (p-list) of the variable symbol. Every atomic symbol has a p-list. When the p-list contains the indicator APVAL, then the symbol is a constant and the next item on the list is the value. eval searches p-lists before a-lists when evaluating variables, thus making it possible to set constants.
- The first pseudo-function is define, it causes new functions to be defined and available within the system. In practice, LABEL is seldom used. It is usually more convenient to attach the name to the definition in a uniform manner. This is done by putting on the property list of the name, the symbolEXPR followed by the function definition. The pseudo-function define accomplishes this. When apply interprets a function represented by an atomic symbol, it searches the p-list of the atomic symbol before searching the current a-list. Thus a define will override a LABEL.
- Another pseudo-function is cset. It is used to create constants. For instance, use cset[X; (A B C D)] to make the variable X always stand for (A B C D).
- Some functions instead of being defined by S-expressions are coded as closed machine language subroutines. Such a function will have the indicator SUBR on its property list followed by a pointer that allows the interpreter to link with the subroutine.
- There are three ways in which a subroutine can be present in the system.

  1. The subroutine is coded into the LISP system.

  2. The function is hand-coded by the user in the assembly type language, LAP.

  3. The function is first defined by an S-expression, and then compiled by the LISP compiler. Compiled functions run from 10 to 100 times as fast as they do when they are interpreted.
- Normally, eval evaluates the arguments of a function before applying the function itself. Thus if eval is given (CONS X Y), it will evaluate X and Y, and then cons them. But if eval is given (QUOTE X), X should not be evaluated. QUOTE is a special form that prevents its argument from being evaluated.
- A special form differs from a function in two ways. Its arguments are not evaluated before the special form sees them. COND, for example, has a very special way of evaluating its arguments by using evcon. The second way which special forms differ from functions is that they may have an indefinite number of arguments. Special forms have indicators on their property lists called FEXPR and FSUBR for LISP-defined forms and machine language coded forms, respectively.

Extension of the LISP Language

- Additions to the LISP Programming System are made to conform to the functional syntax of LISP even though they are not functions. For example, the command to print an S-expression on the output tape is called print. Syntactically, print is a function of one argument. It may be used in composition with other functions, and will be evaluated in the usual manner, with the inside of the composition being evaluated first. It is a function only in the trivial sense that its value happens to be its argument, thus making it an identity function.
- Commands to effect an action such as the operation of input-output, or the defining functions define and cset are called pseudo-functions. It is characteristic of the LISP system that all functions including psuedo-functions must have values.
- Mathematically, it is possible to have functions as arguments of other functions. In LISP, functional arguments are extremely useful. A very important function with a functional argument is maplist.

  ```
  maplist[x; fn] = [null[x] -> NIL; T -> cons[fn[x]; maplist[cdr[x]; fn]]]
  ```

  The functional argument is, of course, a function translated into an S-expression. It is bound to the variable fn and is then used whenever fn is mentioned as a function. The S-expression for maplist itself is as follows:

```
(MAPLIST (LAMBDA (X FN)
            (COND ((NULL X) NIL)
                  (T (CONS (FN X) (MAPLIST (CDR X) FN))) )))
```

- We also need a special rule to translate functional arguments into S-expression. If `fn` is a function used as an argument, then it is translated into `(FUNCTION fn)`.
- The logical or Boolean connectives are usually considered as primitive operators. However, in LISP, they can be defined by using conditional expressions:

  ```
  and[p; q] = [p -> q; T -> F]
  or[p; q] = [p -> T; T -> q]
  not[q] = [q -> F; T -> T]
  ```

  In the System, `not` is a predicate of one argument. However, `and` and `or` are predicates of an indefinite number of arguments, and therefore are special forms.
- In the LISP programming system there are two atomic symbols that represent truth and falsity respectively. These two atomic symbols are `*T*` and `NIL`. It is these symbols rather than `T` and `F` that are the actual value of all predicates in the system. This is mainly a coding convenience. There is no formal distinction between a function and a predicate in LISP. A predicate can be defined as a function whose value is either `*T*` or `NIL`. This is true of all predicates in the System.
- One may use a form that is not a predicate in a location in which a predicate is called for, such as in the `p` position of a `cond` expression, or as an argument of a logical predicate. Semantically, any S-expression that is not `NIL` will be regarded as truth in such a case. One consequence of this is that the predicates `null` and `not` are identical. Another consequence is that `(QUOTE T)` or `(QUOTE X)` is equivalent to `T` as a constant predicate.
- The predicate `eq` has the following behavior.

  1. If its arguments are different, the value of `eq` is `NIL`.

  2. If its arguments are both the same atomic symbol, its value is `*T*`.

  3. If its arguments are both the same, but are not atomic, then the value is `*T*` or `NIL` depending upon whether the arguments are identical in their representation in core memory.

  4. The value of `eq` is always `*T*` or `NIL`. It is never undefined even if its arguments are bad.

LISP List Structures

- The advantages of list structures for the storage of symbolic expressions are:

  1. The size and even the number of expressions with which the program will have to deal cannot be predicted in advance. Therefore, it is difficult to arrange blocks of storage of fixed length to contain them.

  2. Registers can be put back on the free-storage list when they are no longer needed. Even one register returned to the list is of value, but if expressions are stored linearly, it is difficult to make use of blocks of registers of odd sizes that may become available.

  3. An expression that occurs as a subexpression of several expressions need be represented in storage only once.
- Every atomic symbol has a property list. When an atomic symbol is read in for the first time, a property list is created for it. Each property is preceded by an atomic symbol which is called its indicator. Some of the indicators are:

  `PNAME`: the BCD print name of the atomic symbol for input-output use.

  `EXPR`: S-expression defining a function whose name is the atomic symbol on whose property list the EXPR appears.

SUBR: Function defined by a machine language subroutine.

APVAL: Permanent value for the atomic symbol considered as a variable.

The function `get[x;i]` can be used to find a property of `x` whose indicator is `i`.

- The theory of recursive functions developed before will be referred to as elementary LISP. Although this language is universal in terms of computable functions of symbolic expressions, it is not convenient as a programming system without additional tools to increase its power. In particular, elementary LISP has no ability to modify list structure. The only basic function that affects list structure is `cons`, and this does not change existing lists, but creates new lists. LISP is made general in terms of list structure by means of the basic list operators `rplaca` and `rplacd`. These operators can be used to replace the address or decrement or any word in a list. They are used for their effect, as well as for their value, and are also pseudo-functions.
- At any given time only a part of the memory reserved for list structures will actually be in use for storing S-expressions. The remaining registers are arranged in a single list called the free-storage list. A certain register, `FREE`, in the program contains the location of the first register in this list. When a word is required to form some additional list structure, the first word on the free-storage list is taken and the number in register `FREE` is changed to become the location of the second word on the free-storage list. No provision need be made for the user to program the return of registers to the free-storage list. This return takes place automatically whenever the free-storage list has been exhausted during the running of a LISP program. The program that retrieves the storage is called the garbage collector.