LECTURES ON THE LISP INTERPRETER

Paul E. Dawson

Daniel P. Friedman

Computer Science Department

Indiana University

Bloomington, Indiana 47401

TECHNICAL REPORT No. 22

LECTURES ON THE LISP INTERPRETER

PAUL E. DAWSON

DANIEL P. FRIEDMAN

JANUARY, 1975

Lectures on the LISP Interpreter

Paul E. Dawson

Daniel P. Friedman

Computer Science Department

Indiana University

Bloomington, Indiana 47401

Introduction

The subject of these notes is the LISP implementation and discussion of the LISP interpreter as presented in [1]. These notes will provide the reader with a better understanding of LISP, its capabilities and limitations. Through the explanation of the interpreter (see Appendix B in [1]) the reader will better understand such concepts as: variable bindings, evaluation of bindings, function definition, forms, special forms, recursion, and the functional definition of the interpreter procedures.

We assume the reader has a basic working knowledge of LISP and therefore will not attempt to introduce basic LISP concepts

as atomic symbols, S-expressions, list notation, dotted pairs, or the writing of LISP functions.

These notes will first give the reader a broad understanding of the LISP operating environment, function definition using DEFINE and DEFLIST, the use of LAMBDA and LABEL notation, the function of FUNCTION, and the use of PROG before examining the construction of the interpreter itself. Next, by examining the interpreter's construction, we will re-examine these same constructs and others, but this time the emphasis will be on those interpreter components responsible for the recognition and interpretation of these constructs, and how the interpretation is carried out.

This description specifies the effect of the interpreter, not its actual internal workings.

LISP - Overview

The LISP language introduced by McCarthy, et al. in the early 1960's, is a functional list processor designed for symbolic data processing. Its areas of application are that of artificial intelligence and symbolic mathematics. Because LISP is a formal mathematical language, it is of theoretical interest. It is also a programming language of extreme power when used for symbolic calculations in differential and integral calculus, electrical circuit theory, mathematical logic, game playing, particle physics, metacompilers, string transformations, programming language syntax translation, simulations, question/answering systems, linguistics, information retrieval, on-line text editing and formal program language analysis.

All functions and data in LISP are written as symbolic expressions, referred to as S-expressions. That is, functions are written entirely in terms of S-expressions and operate on S-expressions as data. This relationship allows the interpreter or universal processor for the LISP language to be defined as a LISP function.

Variable Bindings - Association List and Property List

A variable is a symbol that is used to represent an argument of a function. Thus we can define the function

$$f(x) = x^3/4$$

where $x = 2$. In this case the answer is 2. The process of arriving at this result involved substituting the number 2 for each occurrence of x in the function. The number 2 is bound to the variable x , or the number 2 is associated with the atomic symbol x .

Constructing the corresponding LISP function

```
(F(LAMBDA(X)
  (COND
  ((ZEROP X) 0 )
  (T(DIVIDE(TIMES X X X)4))  )  ))
```

and executing the function with the call (F 2) , the result is again 2.

The information required to evaluate (F 2) is supplied by the environment. The environment contains all the information which the system processes about each of its atoms. Recall that everything is an atom: function names, operators, variables, and constants. There is no distinction made between atoms. The environment associates atoms with the values they represent.

When the function is applied to its arguments, (F 2) , the existing environment is modified associating bound variables with the function arguments (i.e., X is bound to 2). During the evaluation of the function, the current associations replace any previously existing associations for those variables. After the evaluation has been completed, the associations established by the function invocation are deleted from the environment. Thus the associations of the bound variables are temporary, existing only while the function's form is being evaluated.

In addition to variables, the environment contains constants, primitive functions, and functions introduced by the LISP functions DEFINE and DEFLIST, each having a name and a value. The value is either an atomic symbol or a function.

In [1], the environment is described as an association list. To understand bound variables adequately, we must examine how these associations are constructed.

In interpretive LISP systems, whenever a lambda or program expression is encountered, the variables to be bound are placed on the association list or a-list. The a-list is a LISP list of dotted pairs of the form

$$((u_1 . v_1)(u_2 . v_2) . . . (u_n . v_n))$$

where each of the u_1's is a variable and each of the v_1's are the corresponding values or bindings. Lambda or program variables are paired with their values and the pairs are attached to the front (leftmost) end of the a-list, with previous bindings to the right. In the previous example, the invocation of the function F by (F 2) would cause the variable X to be paired with the

numeric atom 2 and attached to the a-list,

$$((X . 2)(u_2 . v_2) . . . (u_n . v_n)) .$$

During the evaluation of the function, all references to the variable X , are references to the a-list. The a-list is searched from left to right for the first occurrence of the variable X . When it is located, the CDR of the bound pair is returned as the current value of the variable. After the function has been evaluated, the a-list is returned to its previous state by removing the bound variables of that function

$$((u_2 . v_2) . . . (u_n . v_n)) .$$

The original LISP had no constants, so that the single a-list was well suited to the needs of the environment. With the addition of constants, a new type of environment was necessary. A constant must retain its value, regardless of any bindings which may be in effect. In addition to the temporary environment (a-list), a need for a permanent environment arises. Assuming that a permanent environment exists, the interpreter procedures for LISP will first search the permanent environment whenever a variable is evaluated. If the variable has no permanent association of the type being sought, the search proceeds to the temporary environment, the a-list.

The permanent and temporary environments have different practical requirements in terms of access and maintenance. Permanent associations are used often and must be accessed rapidly. They will not be altered frequently, and therefore the updating process need not be highly efficient. Temporary associations, on the other hand, are modified quite frequently and require a more efficient

alteration procedure. The permanent environment is implemented to allow random access to its contents, while the association list is manipulated in a sequential access mode.

The permanent LISP environment is implemented in terms of property lists. Each unique atom within the LISP system has a property list associated with it. The property list of an atom contains the permanent associations of that atom, its "print name," or any constant value. The actual structure of the property list for each atom is behaviorally similar to that of the association list, i.e., bound pairs. The first entity of each pair is an atomic symbol called an indicator. The second entity is an S-expression which represents the value that is associated with the indicator. The temporary environment allows the association of a single value with each bound atom. The permanent environment, on the other hand, allows the programmer to associate multiple values with each atom, in the sense that an atom may have associated with it an indefinite number of indicator-value pairs.

The LISP system provides the user with a number of built-in indicators that have special significance for the interpreter functions. Table 1.1 shows the basic indicators utilized by the LISP interpretive system.

Indicator	Value
PNAME	Atomic print name -- character sequence used to represent atom on an OUTPUT device.
APVAL	Permanent value of an atom. The permanent value of F is NIL.
SUBR	Address of a machine language coded interpreter routine (CAR, CDR, CONS, EQ, ATOM, etc.).
FSUBR	Address of a machine language coded interpreter routine for handling special forms (COND, LIST, QUOTE, OR, AND, TIMES, etc.).
EXPR	User defined function.
FEXPR	User defined function for handling special forms.

Table 1.1

The LISP system provides the programmer with two procedures (SUBR's) for manipulating the property lists of atoms. The LISP procedure (PUT atom indicator value) places the indicator-value pair on the property list of atom. The value of PUT is atom. The LISP procedure (GET atom indicator) retrieves the value that is associated with indicator on the property list of atom. If indicator does not exist on the property list of atom then GET returns NIL, otherwise the value of GET is the stored value.

To see how the environment is used, suppose that the variable X is to be evaluated. First, the property list of X is searched for the indicator APVAL, e.g.,

(GET(QUOTE X)(QUOTE APVAL))

If the value returned by GET is non-NIL, then the CAR of the value is the binding of X . The association list is searched only if the indicator APVAL is not found, and thus an APVAL takes precedence over any other binding.

A variable not bound by the current function definition is a free variable. The binding of a free variable is established outside the current function definition and will be available when that variable is evaluated, either by locating a value on its property list or by locating it on the a-list. The binding of free variables is often set with one of the LISP functions, CSET, CSETQ, SET, or SETQ. Free variables can be used in recursive functions to reduce the length of the association list.

The complete environment consists of the association list and all property lists. Together they provide all the information which the system can process concerning an atom. The organization of the environment is designed to provide rapid, random access to the permanent associations and the most recent temporary bindings.

Function Definition

In LISP, as in other programming languages, we wish to write programs that are parameterized and that compute an answer when values are assigned to the function arguments. However, in LISP we do not use the syntax and program structure of algebraic languages. LISP programs are formulated and written in the mathematical notation of recursive function theory. As such, procedures are functions; parameters are constants, and variables that can be passed between functions are arguments.

Forms and Functions

Given the algebraic expression $X^2 + Y$ evaluate the expression for the values 2 and 3. Immediately there is a notational problem

in that we don't know whether X = 2 and Y = 3 or X = 3 and Y = 2 . To resolve this ambiguity, LISP makes use of Church's lambda notation.

In Church's lambda notation the expression $X^2 + Y$ is called a form. A form is an expression that may be evaluated when a correspondence has been established between its variables and a set of actual arguments. In LISP this formula is represented in Polish prefix notation as

(PLUS(EXPT X 2) Y) .

Furthermore, in Church's lambda notation,

$f = \lambda(x,y)(x^2+y)$

is a function named f , since it satisfies the two necessary conditions for a function:

1. A form to be evaluated
2. A correspondence between the variables of the form and the arguments of the function.

Invoking the function (f 2 3) the previous ambiguity is resolved, since Church's lambda notation provides a correspondence between x , y , and 2, 3, such that (f 2 3) = $2^2 + 3$.

In LISP, (f 2 3) would be defined as

((LAMBDA(X Y)(PLUS(EXPT X 2)Y)) 2 3) .

Lambda expressions consist of three entities:

1. The word LAMBDA.
2. A list of literal atoms that are to be treated as variables (lambda variables).
3. A form to be evaluated.

After recognizing that an S-expression is a lambda expression, the

LISP interpreter pairs the lambda variables with the actual arguments and attaches these pairings to the association list. When the form is evaluated, any reference to a lambda variable causes the interpreter to evaluate that lambda variable by locating it on the association list and substituting its bound value.

More frequently the LISP programmer will use lambda expressions in conjunction with the LISP functions DEFINE and DEFLIST. For example, rather than having to specify the lambda expression everytime it is evaluated with new arguments, the programmer defines the function f as:

```
(DEFINE(QUOTE(
   (F(LAMBDA(X Y)
     (PLUS(EXPT X 2) Y ) ))
     )))
```

and evaluates (F 2 3) as before.

DEFINE is a function of one argument, a list of functions to be "defined." The effect of DEFINE is to place on the property list of the atomic function name the indicator EXPR and the lambda expression as the value. In the previous example, DEFINE would place the indicator EXPR and the value (LAMBDA(X Y)(PLUS (EXPT X 2) Y)) on the property list of F . Now when the programmer executes (F 2 3) , the interpreter locates the lambda expression on the property list of F , (GET (QUOTE F)(QUOTE EXPR)) , binds X to 2 and Y to 3, attaches both of these pairings on the front of the association list and evaluates the form, producing the value 7.

DEFLIST is similar to DEFINE, except that it is more general than DEFINE. DEFLIST is a function of two arguments,

(DEFLIST(QUOTE($f_1 f_2 \ldots f_n$))(QUOTE indicator)) a list of functions to be defined and an indicator to be associated with their definitions on their respective property lists. The uses of DEFLIST will be clarified in the discussion of special forms.

DEFINEing a function allows the storing of a function definition, so that the same function definition may be applied to different arguments without having to redefine the function definition each time it is applied to new arguments.

Earlier we composed and evaluated lambda expressions such as ((LAMBDA(X Y)(PLUS(EXPT X 2)Y)) 2 3) . These were <u>temporary</u> lambda expressions. By naming them, we could make them <u>permanent</u> functions with DEFINE. Recursive expressions point up an inadequacy in lambda notation that requires us to define as permanent, recursive functions that we wish to use as temporary functions. This difficulty results from the inability to call functions recursively since the lambda expression is not named. To resolve this difficulty and allow composition and evaluation of temporary recursive functions LISP provides the LABEL function. To write temporary functions that can call themselves recursively, we write

(LABEL <u>name</u> <u>lambda-expression</u>) .

Example:

```
(LABEL MEMBER(LAMBDA(X SET)(COND
 ((NULL SET)NIL)
 ((EQ X (CAR SET)) T)
 (T(MEMBER X (CDR SET)))  )  ))
```

Label notation creates temporary expressions that may be evaluated in a recursive manner. The labeled lambda expression binds the function name to the lambda expression and attaches the pair to the association list. Labeled lambda expressions may be written

recursively, so that each time the function is referenced internally, its most recent function definition is retrieved from the association list and applied to its argument.

Functions with Functional Arguments

Mathematically, it is possible to have functions as arguments of other functions. In LISP functional arguments are extremely useful and lend themselves to the generality of LISP as a programming language.

When arguments are transmitted to a function, they are evaluated, except when they are transmitted to a function defined as a special form; it controls how its arguments are evaluated. When functions are used as arguments, they should be transmitted unevaluated. The special form FUNCTION is used for this purpose in LISP. FUNCTION acts like QUOTE, and in fact FUNCTION and QUOTE may be used interchangeably, provided there are no free variables present. FUNCTION is used with functional arguments to indicate to the LISP interpreter that a function is being passed as an argument to another function and that its evaluation is to be suppressed. FUNCTION is a special form that takes one argument, a function or lambda expression. It has the form

(FUNCTION fexp)

where fexp is either the name of a previously defined or labeled function, a LISP SUBR or a lambda expression.

An example of the application of functional arguments is the LISP function MAPLIST. MAPLIST is a function of two arguments: an argument list and a function to be applied to the list,

```
(MAPLIST LIST FN)
```

MAPLIST returns as its value a list of the values of the repeated evaluation of FN applied to LIST. The value of MAPLIST may be expressed as

```
(LIST(FN LIST)(FN(CDR LIST))...(FN(CDDD...DR LIST))  .
```

The definition of MAPLIST is

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

Examples:

```
(SQUARE(LAMBDA(L)
    (TIMES(CAR L)(CAR L)))

((LAMBDA(X)(MAPLIST X (FUNCTION SQUARE)))(QUOTE (1 2 3 4 5)))
    = (1 4 9 16 25)

((LAMBDA(X)(MAPLIST X (FUNCTION CDR)))(QUOTE (THIS IS A LIST)))
    = ((IS A LIST)(A LIST)(LIST)())
```

The LISP function FUNCTION allows the programmer to pass, unevaluated, functional arguments to functions. More important and not as apparent, is the relationship between the use of FUNCTION and the state of the environment. The LISP function QUOTE may be used if the net result is to suppress the evaluation of a functional argument. But there arise difficulties relating to the binding of free variables and the use of functional arguments that are not adequately handled by QUOTE. This relationship will be discussed in further detail in the section on the evaluation procedures of the LISP interpreter.

Special Forms

A form is an expression which can be evaluated when some correspondence has been established between the variables contained in it and a set of actual arguments. For instance, (CAR X) is a form.

The procedure for evaluating a form involves evaluating the argument X (i.e., obtaining a value for X from the environment) and then applying the LISP function CAR to the binding of X .

These procedures are followed without regard to the particular variable or function being operated upon. If all LISP forms were evaluated in this manner the capabilities of LISP as a symbolic data processing language would be severely limited. There arises the need for the construction of forms that allow the programmer to:

1. Write forms with an indefinite number of arguments, and/or

2. Write forms for which their arguments are passed unevaluated.

Forms of this nature are called "special forms." Functions can either be built into the LISP system or defined by the programmer. Functions that are DEFINED by the user are called EXPR's (refer to the discussion of property lists) and are characterized as having a fixed number of arguments and the arguments are evaluated prior to calling the function. Functions that are built into the LISP system, such as CAR, CDR, CONS, EQ, and ATOM, are called SUBR's and are likewise characterized by a fixed number of arguments and argument evaluation prior to calling the function. Special forms may either be built into the system or user defined. Each special form is written as a list whose first member is a function name

and whose remaining members are expressions,

$$(\text{func-name } e_1\ e_2\ \ldots\ e_n)$$

Special forms that are built into the LISP system are called FSUBR's and user defined special forms are called FEXPR's.

Special forms, FEXPR's and FSUBR's, are set apart from SUBR's and EXPR's by the manner in which their arguments are handled. Some special forms will utilize both additional capabilities for argument interpretation while others will utilize either one or the other.

LISP programmers, who have used the language to any extent, will recognize that some of the more frequently used functions are implemented as special forms. Some of these special forms (FSUBR's) are:

$$(\text{QUOTE exp})$$
$$(\text{LIST } e_1\ e_2\ \ldots\ e_n)$$
$$(\text{COND } (p_1\ e_1)\ \ldots\ (p_n\ e_n))$$

QUOTE is a special form that receives its single argument unevaluated. The argument may be any S-expression. QUOTE returns the unevaluated S-expression as its value.

LIST is a special form which takes an indefinite number of arguments. The arguments of LIST are not evaluated prior to passing them to the function. The value of LIST is a list of the evaluated arguments.

$$(\text{LIST } e_1\ e_2\ \ldots\ e_n) = (v_1\ v_2\ \ldots\ v_n)$$

where v_i represents the value of e_i .

COND takes an indefinite number of arguments, which are predicate-expression pairs. It accepts these arguments unevaluated;

as a special form it evaluates the predicate of each predicate-expression pair until the first predicate that is non-NIL is found and then the corresponding expression is evaluated and returned as the value of COND.

COND is an ideal example of why special forms are needed in LISP. If the arguments of COND were evaluated before executing COND, the interpreter would be performing a lot of unneeded evaluation, in the sense that there is no need to evaluate $p_{i+1}, p_{i+2} \cdots p_n$ if p_i evaluates as non-NIL. By allowing COND to control the evaluation of its arguments it will call upon the interpreter to perform a minimum of form evaluation.

When a form is encountered the interpreter looks on the property list of the function name for one of the four indicators, SUBR, FSUBR, EXPR, or FEXPR. SUBR and EXPR indicate that the form is a function applied to a fixed-number of evaluated arguments, while FSUBR and FEXPR indicate that it is a special form, written in machine language or LISP respectively, for which the argument list is indefinite in length and that the arguments are passed unevaluated. A user-defined special form (FEXPR) must be placed on the property list of its atomic name with the indicator FEXPR, and hence DEFINE cannot be used. Recall the function DEFLIST is available for placing expressions onto the property list with arbitrary indicators.

EVALQUOTE$

An interpreter or universal function is one that can compute the value of any function applied to its arguments when given a

description of that function.

> An interpreter executes a source-language program by examining the source language and performing the specified algorithm. This is in contrast to a translator or compiler which translates a source-language into machine language for subsequent execution. [1]

The LISP interpreter is a function named EVALQUOTE$. It is applied to two S-expressions from an input medium,

```
EVALQUOTE$( FN ARGS )  .
```

```
(EVALQUOTE$(LAMBDA(FN ARGS)
   (COND
   ((OR(GET FN(QUOTE FEXPR))(GET FN(QUOTE FSUBR)))
        (EVAL$(CONS FN ARGS) NIL ))
   (T(APPLY$ FN ARGS NIL))   )   ))
```

When EVALQUOTE$ is given a function and a list of arguments for that function it computes the value of the function applied to its arguments.

The evaluation procedure for LISP consists of two main functions: the application of a function to its arguments (APPLY$) and the evaluation of a form (EVAL$). APPLY$'s task is to sort out the meaning of the function, find its bound variables, pair them with the function arguments and then hand over the form to EVAL$. When EVAL$ has passed a function, it evaluates the arguments and then calls APPLY$ to bind the variables and to update the environment.

The execution of EVALQUOTE$(FN ARGS) involves deciding whether or not the function FN is a special form. The question is resolved by examining the property list of FN for the indicator FEXPR or FSUBR. If FN is not a special form then EVALQUOTE$ calls APPLY$ with the function, the argument list, and the asso-

ciation list, which is initially NIL. If the function FN is a special form, then EVALQUOTE$ calls EVAL$, a function of two arguments, a form, and the association list. The arguments of EVAL$ are formed by CONSing FN and ARGS to produce a form and an initially empty association list.

The decision to call EVAL$ or APPLY$ is determined by whether the function is a special form. Recall that an indefinite number of arguments and delayed argument evaluation characterize special forms. Since one of the functions of APPLY$ is to decode variable bindings, it should be clear as to why EVAL$ is called when EVALQUOTE$ encounters a special form.

APPLY$ is a function of three arguments: a function FN , and a list of arguments ARGS and the association list, ALIST.

APPLY$ first determines if FN is NIL, and if so APPLY$ will simply return NIL. If FN is non-NIL and FN is an atomic symbol APPLY$ will determine if the function is a user-defined function (i.e., it has the indicator EXPR on its property list) and if so, the value associated with the indicator EXPR is APPLY$'d to ARGS and ALIST (i.e., APPLY$ is called recursively with the definition of FN). If FN is not an EXPR, APPLY$ will determine if FN has been defined as a SUBR. If the function is a SUBR, APPLY$ will be called recursively, but the function to be applied to ARGS will be evaluated,

(APPLY$(EVAL$ FN ALIST)ARGS ALIST) .

If FN is neither an EXPR nor a SUBR then the definition of FN will be determined by examining the association list. If a definition for FN is located on the association list, then this defini-

tion is applied to ARGS:

```
(APPLY$(CDR(SASSOC$ FN ALIST (QUOTE(LAMBDA()
      (ERROR(QUOTE A2)))))) ARGS ALIST )
```

If a definition cannot be located for the atomic symbol FN (i.e., the function definition is not bound in either the permanent environment or the temporary environment) then error A2 is returned, signifying an undefined function.

If it was previously determined that FN was non-atomic then APPLY$ checks if (CAR FN) is the literal atom LABEL. Recall that the use of LABEL notation allows the programmer to define temporary recursively callable functions. The function definition is temporary in that it is not stored in the permanent environment (i.e., on the property list of the atomic function name) but rather the function name is bound to its definition and placed on the association list.

When APPLY$ encounters a labeled function,

```
(EQ(CAR FN)(QUOTE LABEL))
```

it applies the function definition to ARGS and updates the ALIST:

```
(APPLY$(CAR(CDR(CDR FN))) ARGS
    (CONS(CONS(CAR(CDR FN))(CAR(CDR(CDR FN)))) ALIST))
```

```
(APPLY$(LAMBDA(FN ARGS ALIST)
    (COND
    ((NULL$ FN) NIL )
    ((ATOM FN)
        (COND
        ((GET FN(QUOTE EXPR))(APPLY$(GET FN(QUOTE EXPR))ARGS ALIST))
        ((GET FN(QUOTE SUBR))
            (COND
            ((EQ FN(QUOTE DEFINE))(DEFINE$ ARGS))
            ((EQ FN(QUOTE CAR))(CAR(CAR ARGS)))
            ((EQ FN(QUOTE CDR))(CDR(CAR ARGS)))
            ((EQ FN(QUOTE CONS))(CONS(CAR ARGS)(CAR(CDR ARGS))))
            ((EQ FN(QUOTE ATOM))(ATOM(CAR ARGS)))
            ((EQ FN(QUOTE EQ))(EQ(CAR ARGS)(CAR(CDR ARGS))))
            (T(APPLY$(EVAL$ FN ALIST) ARGS ALIST))  )   )
        (T(APPLY$(CDR(SASSOC$ FN ALIST(QUOTE(LAMBDA()(ERROR
            (QUOTE A2)))))) ARGS ALIST))  )   )
    ((EQ(CAR FN)(QUOTE LABEL))(APPLY$(CAR(CDR(CDR FN))) ARGS
            (CONS(CONS(CAR(CDR FN))(CAR(CDR(CDR FN)))) ALIST)))
    ((EQ(CAR FN)(QUOTE FUNARG))(APPLY$(CAR(CDR FN)) ARGS
            (CAR(CDR(CDR FN)))))
    ((EQ(CAR FN)(QUOTE LAMBDA))(EVAL$(CAR(CDR(CDR FN)))
            (NCONC$(PAIR$(CAR(CDR FN))ARGS)ALIST)))
    (T(APPLY$(EVAL$ FN ALIST) ARGS ALIST))  )   ))
```

If (CAR FN) is not equal to LABEL then APPLY$ checks if (CAR FN) is equal to the literal atom FUNARG, (EQ(CAR FN)(QUOTE FUNARG)). An S-expression that has the literal atom FUNARG as its first element is a list that is created when EVAL$ encounters FUNCTION. As stated before FUNCTION is used to pass functional arguments to functions. In order to preserve the environment in which the functional argument was declared, FUNCTION saves the state of the association list at the point at which FUNCTION was encountered by creating the S-expression:

(FUNARG function ALIST) .

When APPLY$ has identified this construct, it will apply the function to ARGS within the environment ALIST:

(APPLY$(CAR(CDR FN)) ARGS (CAR(CDR(CDR FN)))) .

If APPLY$ does not identify either a labeled function or a FUNARG notation, it will try:

```
(EQ(CAR FN)(QUOTE LAMBDA))  ?
```

If APPLY$ encounters a lambda-expression, recalling that a lambda-expression consists of (LAMBDA arg-list form), APPLY$ will evaluate the form, but not before the environment is updated by binding ARGS to arg-list and adding these new bound pairs to the association list!

```
(EVAL$(CAR(CDR(CDR FN)))
      (NCONC$(PAIR$(CAR(CDR FN))ARGS)ALIST))  .
```

Notice that (CAR(CDR(CDR FN))) evaluates to a form and that (CAR(CDR FN)) evaluates to the arg-list.

If APPLY$ cannot recognize any of the possibilities examined so far, then FN is evaluated within the current environment and again applied to ARGS:

```
(APPLY$(EVAL$ FN ALIST) ARGS ALIST )  .
```

EVAL$

EVAL$ evaluates forms using information within the current environment. EVAL$ is a function of two arguments: a form and the association list, FORM and ALIST respectively.

If FORM is NIL then EVAL$ simply returns NIL as its value. If FORM is not NIL then EVAL$ checks if FORM is a numeric atom and if so returns the numeric atom as its value. In other words, LISP numbers evaluate to themselves. If FORM is neither NIL nor a number, EVAL$ determines if the form is an atomic symbol. If this test is true then EVAL$ will first check the permanent environment (i.e., the property list of FORM for the indicator APVAL).

Thus, if (GET FORM(QUOTE APVAL)) is non-NIL, then the CAR of the value is returned as the value of EVAL$.
If FORM has no binding in the permanent environment then the temporary environment, ALIST, is searched. The search of the association list proceeds in a left-to-right manner, returning the value associated with the first occurrence of FORM.

If a binding for FORM cannot be located in either permanent environment or temporary environment then the error A8 is returned, signifying an unbound variable.

If the form to be evaluated is non-atomic then EVAL$ tests if (EQ(CAR FORM)(QUOTE QUOTE)) and if true will simply return (CAR(CDR FORM)). This is in keeping with what has been stated about the LISP special form QUOTE: it simply returns its argument unevaluated.

```
(EVAL$(LAMBDA(FORM ALIST)
   (COND
   ((NULL$ FORM )NIL )
   ((NUMBERP FORM ) FORM )
   ((ATOM FORM)
        (COND
        ((GET FORM(QUOTE APVAL))(CAR(GET FORM(QUOTE APVAL))))
        (T(CDR(SASSOC$ FORM ALIST(QUOTE(LAMBDA()(ERROR
          (QUOTE A8)))))))   )      )
   ((EQ(CAR FORM)(QUOTE QUOTE))(CAR(CDR FORM)))
   ((EQ(CAR FORM)(QUOTE FUNCTION))(LIST(QUOTE FUNARG)(CAR(CDR FORM))
        ALIST))
   ((EQ(CAR FORM)(QUOTE COND))(EVCON$(CDR FORM) ALIST))
   ((EQ(CAR FORM)(QUOTE PROG))(PROG$(CDR FORM) ALIST ))
   ((ATOM(CAR FORM))
        (COND
        ((GET(CAR FORM)(QUOTE EXPR))(APPLY$(GET(CAR FORM)(QUOTE EXPR))
             (EVLIS$(CDR FORM) ALIST) ALIST))
        ((GET(CAR FORM)(QUOTE FEXPR))(APPLY$(GET(CAR FORM)(QUOTE FEXPR
             ))(LIST(CDR FORM) ALIST) ALIST))
        ((GET(CAR FORM)(QUOTE SUBR))
             (COND
             ((EQ(CAR FORM)(QUOTE CAR))(CAR(CAR(EVLIS$(CDR FORM)
             ALIST))))
             ((EQ(CAR FORM)(QUOTE CDR))(CDR(CAR(EVLIS$(CDR FORM)
             ALIST))))
             ((EQ(CAR FORM)(QUOTE CONS))(CONS(CAR(EVLIS$(LIST
             (CAR(CDR FORM)))ALIST))(CAR(EVLIS$(CDR(CDR FORM))
             ALIST))))
             ((EQ(CAR FORM)(QUOTE ATOM))(ATOM(CAR(EVLIS$(CDR FORM)
             ALIST))))
             ((EQ(CAR FORM)(QUOTE EQ))(EQ(CAR(EVLIS$(LIST
             (CAR(CDR FORM)))ALIST))(CAR(EVLIS$(CDR(CDR FORM))
             ALIST))))
             ((EQ(CAR FORM)(QUOTE ERROR))(ERROR(CAR(EVLIS$
             (CDR FORM) ALIST))))
             ((EQ(CAR FORM)(QUOTE NUMBERP))(NUMBERP(CAR(EVLIS$
             (CDR FORM) ALIST))))
             ((EQ(CAR FORM)(QUOTE GET))(GET(CAR(EVLIS$(LIST
             (CAR(CDR FORM)))ALIST))(CAR(EVLIS$(CDR(CDR FORM))
             ALIST))))
             ((EQ(CAR FORM)(QUOTE NCONC ))(NCONC$(CAR(EVLIS$(LIST
             (CAR(CDR FORM)))ALIST))(CAR(EVLIS$(CDR(CDR FORM))
             ALIST))))
             ((EQ(CAR FORM)(QUOTE MAPLIST ))(MAPLIST$(CAR(EVLIS$(LIST
             (CAR(CDR FORM)))ALIST))(CAR(EVLIS$(CDR(CDR FORM))
             ALIST))))
             ((EQ(CAR FORM)(QUOTE EQUAL ))(EQUAL$(CAR(EVLIS$(LIST
             (CAR(CDR FORM)))ALIST))(CAR(EVLIS$(CDR(CDR FORM))
             ALIST))))
             ((EQ(CAR FORM)(QUOTE RETURN)) FORM)
             ((EQ(CAR FORM)(QUOTE NULL ))(NULL$(CAR(EVLIS$(CDR FORM)
             ALIST))))   )      )
        ((GET(CAR FORM)(QUOTE FSUBR))
             (COND
             ((EQ(CAR FORM)(QUOTE LIST))(EVLIS$(CDR FORM)ALIST))
             ((EQ(CAR FORM)(QUOTE SETQ)) FORM )
             ((EQ(CAR FORM)(QUOTE GO)) FORM )
             ((EQ(CAR FORM)(QUOTE OR))(OR$(CDR FORM)))   )   )
        (T(EVAL$(CONS(CDR(SASSOC$(CAR FORM)ALIST(QUOTE(LAMBDA()(ERROR
             (QUOTE A9))))))(CDR FORM))ALIST))   )      ))
   (T(APPLY$(CAR FORM)(EVLIS$(CDR FORM)ALIST) ALIST))   )      ))
```

If EVAL$ does not identify the special form QUOTE, it checks for the special form FUNCTION, (EQ(CAR FORM)(QUOTE FUNCTION))? The usefullness of FUNCTION allows the LISP programmer to pass functions as arguments to other functions unevaluated. It would appear that the use of QUOTE would achieve the same result, i.e., suppressing the evaluation of the argument. The problem with using QUOTE instead of FUNCTION arises when there are free variables present. Recall that the evaluation of a function is dependent on its arguments plus the environment which gives meaning to any variables used by it or any functions that it may call. An important point which must be realized about functional arguments (abbreviated FUNARG's) is that two different environments are involved. The first environment is the one which is in effect when the functional argument is bound as an argument. We will call this one the binding environment. The second environment is the one in effect when the functional argument is activated as a function call. We will call this the activation environment (as in [5]).

Since the binding environment and the activation environment will, in general, differ from each other, it is a nontrivial matter to decide which environment to use in order to evaluate a functional argument. Consider the following example:

```
          (DEFINE (QUOTE (
    (F(LAMBDA(X)
      (COND
      ((ZEROP A) X )
      (T(MINUS X))   )   ))

    (G(LAMBDA(X FUN)
      (PROG()
       (SETQ A 0)
       (RETURN(FUN X)) ) ))      Activation Environment

    (MAIN(LAMBDA(A X FUN)
         (G X FUN)  )   )
)))

(MAIN  1 3 (FUNCTION F) )        Binding Environment
```

Note that the binding environment has A = 1 and that the activation environment has A = 0 . If we use the binding environment to evaluate (F X) then its value will be -3 . If we use the activation environment to evaluate (F X) its value will be 3 . Thus the importance of determining which environment to use should be clear.

From an implementational viewpoint it would be desirable to use the activation environment. But from a programmer's point of view, it will be appropriate to utilize the binding environment.

Consider now what it would require of the LISP system to restore the binding environment for functional arguments. It would require knowing where in the association list the binding environment exists through some pointer to it. Supplying such a mechanism is the function of FUNCTION in LISP. That is, when one transmits a functional argument F , which is to be evaluated in its binding environment, one uses (FUNCTION F) instead of (QUOTE F).

FUNCTION will prevent its argument F from being evaluated, just as QUOTE would. The result of FUNCTION will be an S-expression

which not only contains a reference to F but also a reference to the state of the association list at the point at which free variables were bound. Thus at the time APPLY$ is to apply the function to its arguments, it will be able to use the binding environment. When EVAL$ encounters FUNCTION, (EQ(CAR FORM)(QUOTE FUNCTION)) where FORM = (FUNCTION F) it returns as its value a list:

```
(LIST(QUOTE FUNARG)(CAR(CDR FORM))ALIST) =
(FUNARG F ALIST)  .
```

At this point let us re-examine the steps taken by APPLY$ when it encounters FUNARG. When APPLY$ checks, (EQ(CAR FN)(QUOTE FUNARG)), where FN = (FUNARG F ALIST), it will apply the function F to ARGS, within the binding environment:

```
((EQ(CAR FN)(QUOTE FUNARG))
   (APPLY$(CAR(CDR FN)) ARGS (CAR(CDR(CDR FN)))))
```

therefore achieving the desired result.

If (CAR FORM) is not FUNCTION then EVAL$ checks, (EQ(CAR FORM)(QUOTE COND))? If EVAL$ encounters a form structured as

$$(\text{COND } (p_1\ e_1)(p_2\ e_2) \ldots (p_n\ e_n)) \ ,$$

where p_i , i=1 , n are predicates and e_i , i=1 , n are expressions, then EVAL$ calls upon another of the interpreter functions, EVCON$ (EValuate CONditional). The value of EVCON$, as a result of evaluating the conditional expression, will be the value returned by EVAL$.

If the test for COND fails then EVAL$ will determine if the form is a PROG definition, (EQ(CAR FORM)(QUOTE PROG)). If the result of this predicate is non-NIL then EVAL$ calls upon another interpreter function PROG$.

Now that EVAL$ has determined that (CAR FORM) is not QUOTE, FUNCTION, COND or PROG, it will check to see if (CAR FORM) is an atomic symbol, (ATOM(CAR FORM))? If it is an atomic symbol there are four cases to consider. First EVAL$ checks to see if the atomic symbol has the indicator EXPR on its property list. For example, if we had DEFINE'd a function TEST which takes two arguments, then EVAL$ would evaluate the form, (TEST(CAR X)(CONS X Y)), where (CAR X) provides the first argument and (CONS X Y) the second argument. The predicate (ATOM(CAR FORM)) will return non-NIL, therefore EVAL$ tests (GET(CAR FORM)(QUOTE EXPR)). Since DEFINE was used to define the function TEST this predicate will likewise return true, and the lambda-expression of TEST will be applied to its arguments:

```
(APPLY$(GET(CAR FORM)(QUOTE EXPR))
       (EVLIS$(CDR FORM)ALIST)  ALIST )  ,
```

where

```
(GET(CAR FORM)(QUOTE EXPR)) = lambda-expression
(EVLIS$(CDR FORM)ALIST)     = list of evaluated  arguments
ALIST                       = association list
```

Recall that functions defined as EXPR's expect their arguments evaluated, and this is the purpose of EVLIS$. EVLIS$ is similar to EVAL$, but unlike EVAL$ which evaluates a <u>single</u> form, EVLIS$ will evaluate a <u>list</u> of forms. With this in mind we can see in the example that,

```
(EVLIS$(CDR FORM) ALIST ) =
(EVLIS$( (CAR X)(CONS X Y) ) ALIST )  .
```

EVLIS$ will evaluate the list of forms, ((CAR X)(CONS X Y)) within the current state of the environment and return a list representing the values of these forms. If the current state of the association list is represented by

((X.(A B C))(Y.(D E)))

then the value returned by EVLIS$ would be the list,

(A ((A B C) D E)) ,

which are the arguments to which the lambda-expression associated with TEST is applied.

If the atomic symbol (CAR FORM) has no value associated with the indicator EXPR, then EVAL$ will determine if the atomic symbol has been defined as a special form (i.e., the indicator FEXPR has a non-NIL value associated with it). If it does, then EVAL$ will do essentially the same thing as it did when it encountered an EXPR, but it will apply the function definition to the unevaluated arguments. In the case of an EXPR, EVAL$ called upon EVLIS$ to evaluate the list of arguments before applying the function to its arguments. With the case of a FEXPR, EVAL$ will perform the following:

(APPLY GET(CAR FORM)(QUOTE FEXPR))(LIST(CDR FORM)ALIST)ALIST) .

Notice that (CDR FORM), which represents the list of arguments, is passed to APPLY$ unevaluated.

If in the last example the function TEST had been defined as a FEXPR and EVAL$ encountered the form (TEST(CAR X)(CONS X Y)) then the list (((CAR X)(CONS X Y))) provides the list of arguments (for FEXPR's always two) to which the definition of TEST is applied.

If the atomic symbol (CAR FORM) has neither the indicators EXPR or FEXPR on its property list then EVAL$ looks to see if it is a SUBR, and if so it will determine which of a predefined set of SUBR's it is. Once EVAL$ has identified the correct SUBR it performs the required algorithm and returns the result of the algorithm as its value.

If EVAL$ has not been able to locate any of the four indicators, EXPR, FEXPR, SUBR, or FSUBR, on the property list of the atomic symbol, then it will try to locate a binding for the atomic symbol on the association list and re-evaluate the new form, created by CONS'ing the binding of (CAR FORM) onto (CDR FORM):

```
(EVAL$(CONS(CDR(SASSOC$(CAR FORM) ALIST (QUOTE (LAMBDA()
      (ERROR(QUOTE A9))))))(CDR FORM)) ALIST)  .
```

If a binding for (CAR FORM) cannot be found on the association list, then error A9 is returned, signifying an undefined function.

if in the previous example we had defined TEST with the LISP LABEL notation, then we know that the functional definition of TEST is not stored in the permanent environment, but rather the definition of TEST is bound to the atomic symbol TEST and placed on the association list. When EVAL$ encounters the form (TEST(CAR X)(CONS X Y)) it will search the association list for the atomic symbol TEST, then CONS its binding onto (CDR FORM) to produce a form to be evaluated.

```
(EVAL$(CONS(CDR(SASSOC$(CAR FORM) ALIST(QUOTE(LAMBDA()
      (ERROR(QUOTE A9))))))(CDR FORM)) ALIST )

= (EVAL$(CONS(function-definition of TEST)
      ((CAR X)(CONS X Y)) ) ALIST )

= (EVAL$(function-definition of TEST(CAR X)(CONS X Y))ALIST)
```

Notice that because EVAL$ examines the property list of an atomic function name for EXPR or FEXPR before it checks for a SUBR or FSUBR the LISP programmer may redefine LISP primitives. For example, the programmer may define the function NULL, which is a SUBR, and have his definition override the host system definition.

Also, notice that the order in which forms are evaluated is consistent with earlier statements, in that the permanent environment is examined for a variable binding before the temporary environment.

When EVAL$ tests (ATOM(CAR FORM)), and the predicate returns NIL, then (CAR FORM) will be applied to the evaluated list, (CDR FORM):

(APPLY$(CAR FORM)(EVLIS$(CDR FORM)ALIST)ALIST) .

This concludes the discussion of the three primary interpreter functions: EVALQUOTE$, APPLY$, and EVAL$. The remaining discussion in this section is concerned with an elaboration of the interpreter's secondary functions: EVCON$, EVLIS$, SASSOC$, PAIR$, and PROG$.

EVCON$

EVCON$ is a LISP function of two arguments, a list of predicate-expression pairs and the association list. EVCON$ evaluates CONDitional expressions.

When EVAL$ encounters the form,

(COND(p_1 e_1)(p_2 e_2) . . . (p_n e_n))

it calls EVCON$ with the CDR of the form and the ALIST. Earlier it was stated that COND is a special form, whose function it is to evaluate each of the predicates, p_i , until one of them returns a non-NIL value and then returns the value resulting from the evaluation of the corresponding e_i .

EVCON$ is a function of two arguments, CONDITION and ALIST. EVCON$ first checks if CONDITION is NIL (i.e., if all of the predicate-expression pairs have been exhausted), and if so, returns

error A3, signifying the value of COND is undefined. If CONDITION is non-NIL, then EVCON$ will evaluate the first predicate by calling EVAL$, with the predicate as the form to be evaluated and the current association list:

(EVAL$(CAR(CAR CONDITION)) ALIST) .

If the evaluation of the predicate returns a non-NIL value, then EVCON$ evaluates and returns as its value the corresponding expression:

(EVAL$(CAR(CDR(CAR CONDITION))) ALIST) .

If the evaluation of the predicate returns NIL then EVCON$ proceeds by recursing with (CDR CONDITION) and ALIST, therefore eliminating the predicate-expression pair that was just examined. Example:

```
(EVCON$(LAMBDA(CONDITION ALIST)
   (COND
   ((NULL$ CONDITION)(ERROR(QUOTE A3)))
   ((EVAL$(CAR(CAR CONDITION))ALIST)(EVAL$(CAR(CDR(CAR CONDITION)))
         ALIST))
   (T(EVCON$(CDR CONDITION) ALIST))   )     ))
```

EVLIS$

In the discussion of EVAL$, it was stated that EVAL$ evaluates a form and that EVLIS$ evaluates a list of forms, returning as its value a list of the values of the evaluated forms.

A logical way to view the execution of EVLIS$ is that it makes repeated calls to EVAL$, each time using a different member of the list of forms and returns a list of the values.

This is in fact the manner in which EVLIS$ is implemented. We may express the execution of EVLIS$ as:

(EVLIS$ ($form_1$ $form_2$ . . . $form_n$) ALIST) =
((EVAL$ $form_1$ ALIST)(EVAL$ $form_2$ ALIST)...(EVAL$ $form_n$ ALIST)) .

In order to perform this mapping, EVLIS$ makes use of the LISP function MAPLIST$. MAPLIST$ is a function of two arguments, a list ℓ , and a function f , to be applied to the list. MAPLIST$ is a mapping of the list ℓ onto the new list (f ℓ) .

To evaluate a list of forms, EVLIS$ is written as

```
(EVLIS$(LAMBDA(ELIST ALIST)
   (MAPLIST$ ELIST (QUOTE(LAMBDA(ARG)(EVAL$(CAR ARG) ALIST))))  ))
```

Example:

```
(EVLIS$ ((CAR X)(CONS X Y)) ( (X.(A B))(Y.(C D)) ) ) =
( A ((A B) C D) )
```

SASSOC$

The LISP function SASSOC$ provides the interpreter with the means of evaluating variable bindings within the temporary environment.

SASSOC$ is a function of three arugments: an atomic symbol to be evaluated, FX , the association list, ALIST , and an error function, ERRFUN , to be executed if the atomic symbol does not have a binding on the association list.

Variable bindings are stored on the association list as variable-value pairs. SASSOC$ first checks if the association list is empty, and if so, it executes ERRFUN, a function of zero arguments.

If the association list is not empty then SASSOC$ checks if the variable of the first variable-value pair is equal to the atomic symbol that is being evaluated:

```
(EQ(CAR(CAR ALIST)) FX ) ?
```

If the predicate returns non-NIL then SASSOC$ returns as its value the variable-value pair (CAR ALIST). If the test for EQuality returns NIL, then SASSOC$ recurses with the CDR of ALIST in order to consider the remaining variable-value pairs. Example:

```
(SASSOC$ X ( (Y.(A B))(X.(D T)))(LAMBDA()(ERROR(QUOTE A8))) ) =
(X D T)

(SASSOC$(LAMBDA(FX ALIST ERRFUN)
   (COND
   ((NULL$ ALIST)(ERRFUN))
   ((EQ(CAR(CAR ALIST)) FX)(CAR ALIST))
   (T(SASSOC$ FX (CDR ALIST) ERRFUN))   )     ))
```

PAIR$

PAIR$ is the LISP function that is called upon to create variable-value pairs.

PAIR$ takes as its arguments two lists. The first list is a list of variables and the second is a list of corresponding values.

PAIR$ CONS's each variable of the first list onto the corresponding value within the second list, returning as its value a list composed of variable-value pairs. If the variable list is longer than the value list then error F3 is returned, and if the value list is longer than the variable list then error F2 is returned. Example:

```
(PAIR$ (X Y Z) (A B C) ) = ( (Z . C) (Y . B) (X . A) )
(PAIR$ (X Y) ((A B C) D)) ) =  ( (Y . D) (X . (A B C)) )
```

```
(PAIR$(LAMBDA(ARG1 ARG2)
   (PROG(A1 A2 PLIST)
        (SETQ A1 ARG1)
        (SETQ A2 ARG2)
   AA   (COND
        ((NULL$ A1)(COND
                   ((NULL$ A2)(RETURN PLIST))
                   (T(ERROR(QUOTE F2)))  )  )
        ((NULL$ A2)(ERROR(QUOTE F3)))       )
        (SETQ PLIST (CONS(CONS(CAR A1)(CAR A2)) PLIST))
        (SETQ A1 (CDR A1))
        (SETQ A2 (CDR A2))
        (GO AA)  )      ))
```

PROG$

PROG$ is a function of two arguments. It is called when EVAL$ encounters the form

(PROG(prog-var-list) exp_1 exp_2 . . . exp_n) .

PROG$ is called with the CDR of the form and the association list, PBODY and ALIST respectively.

PROG$ is written as a program expression with the PROG variables B, A, GLIST, BCAR, and TEMP. Upon entering PROG$, B is bound to PBODY, A is bound to an updated association list in which each of the PROG variables is initially bound to NIL and GLIST is bound to the value returned by (GOLIST$(CDR B)).

GOLIST$ takes as its argument the list of expressions (exp_i's) that constitute the program definition. It searches the list for atomic symbols, which are understood to be PROG labels, and binds each label to the CDR of the list at the point it was encountered. The value returned by GOLIST$ is a list of pairs in which each PROG label is bound to a portion of the PROG definition.

At the label L1, B is bound to (CDR B), eliminating the prog-var-list.

At L2, BCAR is bound to the CAR of B, the next S-expression (of the PROG-body) to be evaluated.

Beginning at L3, if BCAR is an atomic symbol, it is interpreted as a PROG label, and is bypassed by executing (GO L1).

If BCAR is of the form (SETQ var exp) the name of the variable is located on the association list and its value is replaced with the evaluation of exp.

If (CAR BCAR) is the atomic symbol GO, indicating BCAR is the form (GO label), then B is bound to the value returned by calling SASSOC$ with label and GLIST. B is bound to that portion of the PROG-body resulting from a PROG transfer. If a transfer is made to a nonexistent label, then label A6 is returned.

When the form (RETURN exp) is encountered the host LISP function is called with the value of exp.

If BCAR has not been recognized as one of the forms considered, then it is evaluated within the current environment, (EVAL$ BCAR A). If the value of BCAR is not a form using GO, SETQ, or RETURN, then its value is ignored and a transfer is made to L1, eliminating the form from further consideration.

Miscellaneous Help Functions

(NCONC$ ARG1 ARG2)	concatenates its arguments without copying the first one. It changes existing list structure.
(EQUAL$ ARG1 ARG2)	this is a predicate that is true if its arguments are identical S-expressions, and is false if they are different.

(DEFINE$ ARGLIST)	this function takes as its argument a list of functions to be defined. It places on the property list of each atomic function name the indicator EXPR and the function definition as the value.
(REPLACE$ SUB OBJ ALIST)	locates SUB on the association list and replaces its current binding with OBJ.
(NLIST$ NUM)	creates a list of NUM elements where each element is the atom NIL.
(OR$ ORL ALIST)	evaluates each expression in the list of expression ORL from left to right, until one is found that is true, or until the end of the list is reached. The value of OR$ is true or false respectively.
(NULL$ ARG)	determines if its evaluated argument is the atom NIL. If so, then it returns true else false.

```
(PROG$(LAMBDA(PBODY ALIST)
 (PROG(B A GLIST BCAR TEMP)
    (SETQ B PBODY)
    (SETQ A(NCONC$(PAIRS(CAR B)(NLIST$(LENGTH$(CAR B))))) ALIST ))
    (SETQ GLIST (GLIST$(CDR B)))
 L1   (SETQ B(CDR B))
 L2   (SETQ BCAR(CAR B))
 L3   (COND
        ((NULL$ BCAR)(ERROR(QUOTE A3)))
        ((ATOM BCAR)(GO L1))
        ((EQ(CAR BCAR)(QUOTE SETQ))
          (SETQ A(REPLACE$(CAR(CDR BCAR))
          (EVAL$(CAR(CDR(CDR BCAR)))A)A))
          (GO L1) )
        ((EQ(CAR BCAR)(QUOTE GO))
          (SETQ B(CDR(ASSOC$(CAR(CDR BCAR)) GLIST
          (QUOTE(LAMBDA)((ERROR(QUOTE A5)))))))
          (GO L2) )
        ((EQ(CAR BCAR)(QUOTE RETURN))
          (RETURN(EVAL$(CAR(CDR BCAR))A)) )
        (T(SETQ TEMP(EVAL$ BCAR A))
          (COND
          ((NOT(ATOM TEMP))
             (COND
             ((OR$(EQ(CAR TEMP)(QUOTE SETQ))
                   ((EQ(CAR TEMP)(QUOTE GO))
                    ((EQ(CAR TEMP)(QUOTE RETURN)) )
                      (SETQ BCAR TEMP)(GO L3) )
             (T(GO L1)))) )
          (T(GO L1)))))) )) ))
```

Conclusion

We have presented a tutorial on the LISP 1.5 interpreter (see Appendix for fully running version) as originally developed by McCarthy, et al. As with other programming languages, LISP is endowed with specific programming capabilities which lend itself to certain programming applications.

By focussing on the mechanics of the interpreter rather than its underlying philosophy, it is hoped that all LISP programmers will develop an understanding of interpretive languages in general, but more specifically they will become better LISP programmers.

We have presented the interpreter by stating its LISP definition. By defining the LISP interpreter in LISP we have tried to bring to focus the power of the language while preserving its simple but elegant methods.

Within the mechanics of the interpreter lies the operational distinction between the two data environments established, maintained, and utilized by the LISP system. Understanding these differences will lead to a more efficient means of representing data within LISP.

The definition of the LISP interpreter is not complete in that its operational capabilities may be increased by adding to its coded definition. The reader is encouraged to do so, thereby increasing his inderstanding of its definition.

call by value → EXPR passing
name → FEXPR — but args have to be quoted.

References

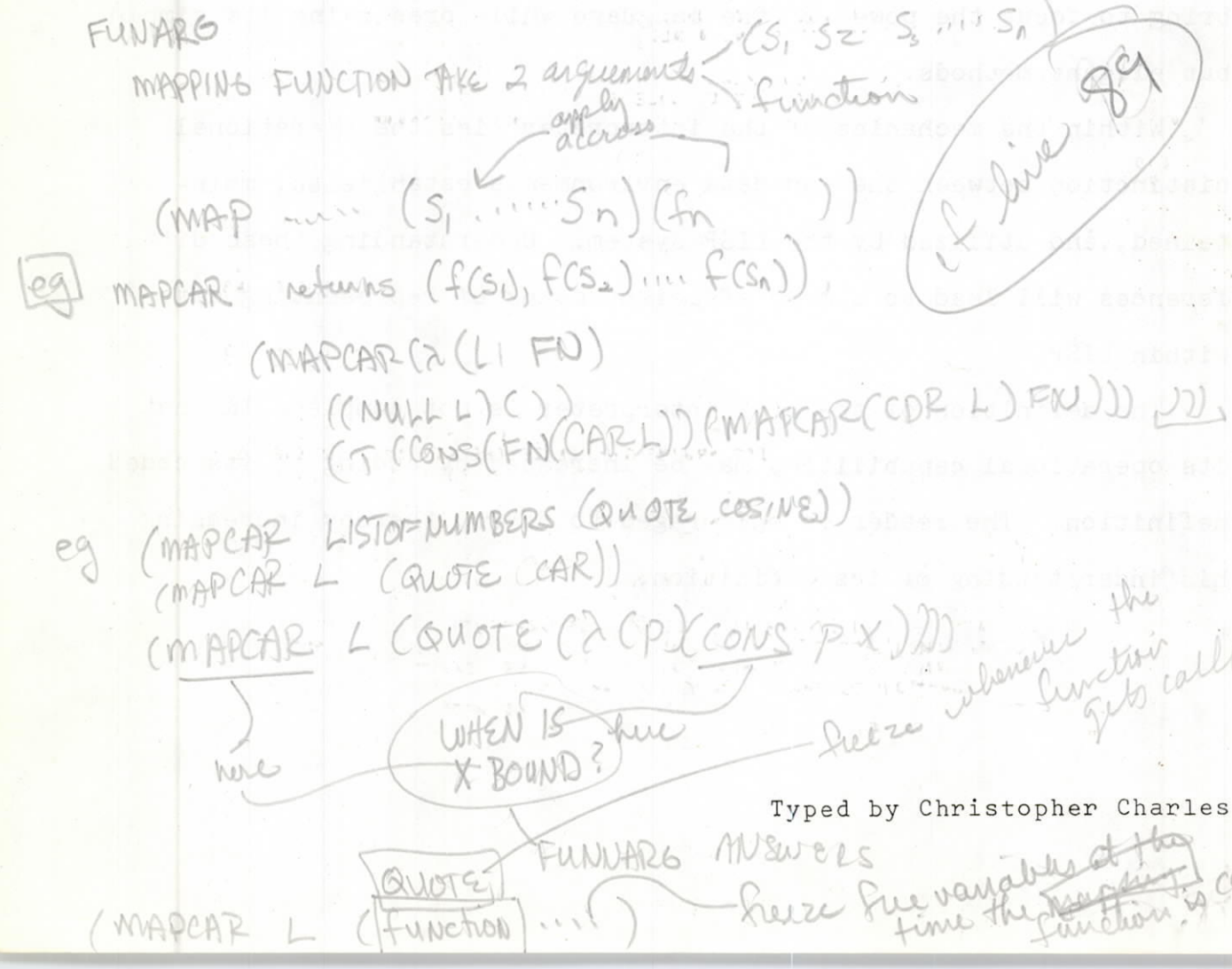
1. McCarthy, J; Abrahams, P.W.; Edwards, D.J.; Hart, T.P.; and Levin, M.I. LISP 1.5 Programmer's Manual, MIT Press, Cambridge, Massachusetts (1962).

2. Weissman, Clark. LISP 1.5 Primer, Dickenson Pub. Co., Belmont, California (1967).

3. Waite, William M. Implementing Software for Non-Numeric Applications, Prentice-Hall, Inc., Englewood, California (1973).

4. Cohen, C., and Zuckerman, C. Evalquote in simple FORTRAN: a tutorial on interpreting LISP. Bit 12 (1972), 299-317.

5. Moses, Joel. The function of FUNCTION in LISP. MIT Project MAC, Cambridge, Massachusetts (June, 1970).



Typed by Christopher Charles

```
?*************************************************************************?LISP   2
?*************************************************************************?LISP   3
?                                                                         ?LISP   4
?                  L I S P   I N T E R P R E T E R S                      ?LISP   5
?                                                                         ?LISP   6
?*************************************************************************?LISP   7
?                                                                         ?LISP   8
?                                  BY                                     ?LISP   9
?                              PAUL E DAWSON                              ?LISP  10
?                                                                         ?LISP  11
?                           INDIANA UNIVERSITY                            ?LISP  12
?                        DEPARTMENT OF COMPUTER SCIENCE                   ?LISP  13
?                         BLOOMINGTON, INDIANA  47401                     ?LISP  14
?                                                                         ?LISP  15
?                              MAY 1974                                   ?LISP  16
?                                                                         ?LISP  17
?*************************************************************************?LISP  18
(DEFINE(QUOTE(                                                             LISP  19
?*************************************************************************?LISP  20
?                                                                         ?LISP  21
?                        E V A L Q U O T E $                              ?LISP  22
?                                                                         ?LISP  23
?-------------------------------------------------------------------------?LISP  24
?     EVALQUOTE$ IS THE TOPLEVEL FUNCTION OF THE LISP INTERPRETER.        ?LISP  25
?   IT TAKES AS ITS ARGUMENTS A FUNCTION AND A LIST OF ARGUMENTS FOR      ?LISP  26
?   THAT FUNCTION.  IF THE FUNCTION HAS BEEN DEFINED AS A "SPECIAL        ?LISP  27
?   FORM" (I.E. IT WAS DEFINED AS A FEXPR OR FSUBR) THEN THE FUNCTION     ?LISP  28
?   AND ITS ARGUMENTS AS CONS") TOGETHER AND PASSED TO EVAL$ WITH A       ?LISP  29
?   NULL ASSOCIATION LIST.  IF THE FUNCTION IS NOT A "SPECIAL FORM"       ?LISP  30
?   THEN THE FUNCTION, ITS ARGUEMTNS AND A NULL ASSOCIATION LIST          ?LISP  31
?   ARE PASSED TO APPLY$.  NOTE THAT IF THE FUNCTION IS A SPECIAL FORM    ?LISP  32
?   ITS ARGUMENTS ARE NOT EVALUATED PRIOR TO PASSING THEM TO EVAL$        ?LISP  33
?   AND THE LENGTH OF THE ARGUMENT LIST IS UNDETERMINED.                  ?LISP  34
?*************************************************************************?LISP  35
 (EVALQUOTE$(LAMBDA(FN ARGS)                                               LISP  36
    (COND                                                                  LISP  37
    ((OR(GET FN(QUOTE FEXPR))(GET FN(QUOTE FSUBR)))                        LISP  38
        (EVAL$(CONS FN ARGS) NIL ))                                        LISP  39
    (T(APPLY$ FN ARGS NIL))   )     ))                                     LISP  40
?*************************************************************************?LISP  41
?                                                                         ?LISP  42
?                           A P P L Y $                                   ?LISP  43
?                                                                         ?LISP  44
?-------------------------------------------------------------------------?LISP  45
?     APPLY$ APPLIES A FUNCTION TO ITS ARGUEMNTS.  THE FIRST              ?LISP  46
?   ARGUMENT OF APPLY$ IS A FUNCTION.  IF IT IS AN ATOMIC                 ?LISP  47
?   SYMBOL, THEN APPLY$ CHECKS (1) TO SEE IF THE FUNCTION HAS BEEN        ?LISP  48
?   DEFINED AS AN EXPR, IF SO THE PROPERTY ASSOCIATED WITH THE            ?LISP  49
?   INDICATOR "EXPR" IS RETRIEVED FROM THE PROPERTY LIST OF THAT          ?LISP  50
?   ATOMIC SYMBOL AND IS APPLIED TO ITS ARGUMENT, (2) IF THE ATOMIC       ?LISP  51
?   FUNCTION SYMBOL HAS BEEN DEFINED AS A SUBR, THEN THE APPROPRIATE      ?LISP  52
?   FUNCTION IS APPLIED TO THE ARGUMENTS, OR (3) IF IT IS NEITHER         ?LISP  53
?   A SUBR OR EXPR THEN ITS MEANING/DEFINITION MUST BE LOOKED UP ON       ?LISP  54
?   THE ASSOCIATION LIST AND THEN APPLIED TO THE ARGUMENTS.               ?LISP  55
?     IF THE FUNCTION ARGUMENT IS NOT ATOMIC AND IF THE FIRST ELEMENT     ?LISP  56
?   OF THE LIST IS "LAMBDA", THEN THE ARGUMENTS ARE PAIRED WITH           ?LISP  57
?   THEIR BOUND VARIABLES (THE ARGUMENTS AND THEIR BINDINGS ARE           ?LISP  58
?   PLACED ON THE ASSOCIATION LIST), AND THE FORM IS GIVEN TO EVAL$       ?LISP  59
?   TO EVALUATE.  IF THE FIRST ELEMENT OF THE LIST IS "LABEL", THEN       ?LISP  60
?   THE FUNCTION NAME AND DEFINITION ARE ADDED TO THE ASSOCIATION LIST    ?LISP  61
?   AND THE INSIDE FUNCTION IS EVALUATED BY APPLY$.  IF THE FIRST         ?LISP  62
?   ELEMENT OF THE LIST IS "FUNARG", THEN THE ELEMENT FOLLOWING           ?LISP  63
?   "FUNARG" IS APPLIED TO THE ARGUMENTS.                                 ?LISP  64
```

```
?     IF NONE OF THESE CONDITIONS ARE MET THEN THE FUNCTION IS          ?LISP  65
?   EVALUATED BY EVAL$ AND REAPPLIED TO ITS ARGUMENTS.                  ?LISP  66
?     ***NOTE*** - APPLY$ HAS NO MECHANISM FOR HANDLING SPECIAL         ?LISP  67
?   FORMS.                                                              ?LISP  68
?*******************************************************************?LISP  69
(APPLY$(LAMBDA(FN ARGS ALIST)                                          LISP  70
    (COND                                                              LISP  71
    ((NULL$ FN) NIL )                                                  LISP  72
    ((ATOM FN)                                                         LISP  73
         (COND                                                         LISP  74
         ((GET FN(QUOTE EXPR))(APPLY$(GET FN(QUOTE EXPR))ARGS ALIST))  LISP  75
         ((GET FN(QUOTE SUBR))                                         LISP  76
            (COND                                                      LISP  77
            ((EQ FN(QUOTE DEFINE))(DEFINE$ ARGS))                      LISP  78
            ((EQ FN(QUOTE CAR))(CAR(CAR ARGS)))                        LISP  79
            ((EQ FN(QUOTE CDR))(CDR(CAR ARGS)))                        LISP  80
            ((EQ FN(QUOTE CONS))(CONS(CAR ARGS)(CAR(CDR ARGS))))       LISP  81
            ((EQ FN(QUOTE ATOM))(ATOM(CAR ARGS)))                      LISP  82
            ((EQ FN(QUOTE EQ))(EQ(CAR ARGS)(CAR(CDR ARGS))))           LISP  83
            (T(APPLY$(EVAL$ FN ALIST) ARGS ALIST))    )     )          LISP  84
        (T(APPLY$(CDR(SASSOC$ FN A_IST(QUOTE(LAMBDA))(ERROR            LISP  85
          (QUOTE A2))))) ARGS ALIST))    )     )                       LISP  86
    ((EQ(CAR FN)(QUOTE LABEL))(APPLY$(CAR(CDR(CDR FN))) ARGS           LISP  87
         (CONS(CONS(CAR(CDR FN))(CAR(CDR(CDR FN))))) ALIST)))          LISP  88
    ((EQ(CAR FN)(QUOTE FUNARG))(APPLY$(CAR(CDR FN)) ARGS               LISP  89
         (CAR(CDR(CDR FN))))))                                         LISP  90
    ((EQ(CAR FN)(QUOTE LAMBDA))(EVAL$(CAR(CDR(CDR FN)))               LISP  91
         (NCONC$(PAIR$(CAR(CDR FN))(ARGS)ALIST)))                      LISP  92
    (T(APPLY$(EVAL$ FN ALIST) ARGS ALIST))    )     ))                 LISP  93
?*******************************************************************?LISP  94
?*******************************************************************?LISP  95
?                                                                     ?LISP  96
?                              E V A L $                              ?LISP  97
?                                                                     ?LISP  98
?-------------------------------------------------------------------?LISP  99
?     EVAL$ EVALUATES FORMS. IT TAKES AS ITS ARGUMENTS A FORM AND THE ?LISP 100
?   ASSOCIATION LIST.  IF THE FORM IS A NUMERIC ATOM THEN THE NUMBER   ?LISP 101
?   IS RETURNED AS THE VALUE OF EVAL$.  IF THE FORM IS ATOMIC AND NOT  ?LISP 102
?   NUMERIC THEN ITS PROPERTY LIST IS CHECKED FOR THE INDICATOR        ?LISP 103
?   'APVAL' AND IF PRESENT THEN THE PERMANENT BINDING IS RETURNED.     ?LISP 104
?   IF THERE IS NO INDICATOR 'APVAL' ON THE PROPERTY LIST THEN THE     ?LISP 105
?   ATOMIC SYMBOL MUST BE A VARIABLE AND ITS BINDING IS FOUND ON THE   ?LISP 106
?   ASSOCIATION LIST.                                                  ?LISP 107
?     IF THE CAR OF THE FORM IS 'QUOTE', THEN IT IS A CONSTANT         ?LISP 108
?   AND THE VALUE RETURNED IS THE CADR OF THE FORM.  IF THE CAR OF THE ?LISP 109
?   FORM IS 'COND', THEN IT IS A CONDITIONAL EXPRESSION AND EVCON$     ?LISP 110
?   EVALUATES THE PROPOSITIONAL TERMS IS ORDER AND RETURNS THE FORM    ?LISP 111
?   FOLLOWING THE FIRST TRUE PREDICATE.                                ?LISP 112
?     IF THE CAR OF THE FORM IS ATOMIC THEN EVAL$ CHECKS TO SEE IF     ?LISP 113
?   IT HAS BEEN DEFINED AS A FEXPR, EXPR, SUBR OR FSUBR.  IF IT IS AN  ?LISP 114
?   EXPR THEN ITS LAMBDA DEFINITION IS APPLIED TO ITS EVALUATED)       ?LISP 115
?   ARGUMENTS.  IF IT IS A FEXPR THEN ITS LAMBDA DEFINITION IS APPLIED ?LISP 116
?   TO ITS UNEVALUATED ARGUMENTS.  IF IT IS A SUBR OR FSUBR THEN THE   ?LISP 117
?   APPROPRIATE OPERATION IS PERFORMED.  IF THE CAR OF THE FORM IS NOT ?LISP 118
?   AN EXPR, FEXPR, SUBR OR FSUBR THEN ITS BINDING IS FOUND ON THE     ?LISP 119
?   ASSOCIATION LIST AND EVALUATED.                                    ?LISP 120
?*******************************************************************?LISP 121
(EVAL$(LAMBDA(FORM ALIST)                                              LISP 122
    (COND                                                              LISP 123
    ((NULL$ FORM )NIL )                                                LISP 124
    ((NUMBERP FORM ) FORM )                                            LISP 125
    ((ATOM FORM)                                                       LISP 126
         (COND                                                         LISP 127
         ((GET FORM(QUOTE APVAL))(CAR(GET FORM(QUOTE APVAL))))         LISP 128
         (T(CDR(SASSOC$ FORM ALIST(QUOTE(LAMBDA))(ERROR                LISP 129
           (QUOTE A8))))))    )     )                                 LISP 130
```

BETTER BE EXPR OR SUBR

?

some loop

test

λ(arglist) (function)

which consists of the pairs created by mapping

NEW ALIST FOR EVAL

λ(— —)

A: → Args

eg. (FUNCTION (λ (A) (CONS A A))) returns (FUNARG (λ ...) ((→ (—) :C)))
(COND (.....)) returns ((( ) ALIST)

SUBR ~ EXPR IN THAT its arguments are evaluated

EVLIST TAKES A LIST OF UNEVALUATED FORMS, SEARCHES THE A-LIST AND RETURNS THE BINDINGS.

eg (CONS A L)

A (CAR (CDR FORM))

(A) (LIST (CAR (CDR FORM)))

(X) (EVLIS (LIST (CADR FORM)))

(CAR FORM is not ATOM) could be a λ-exp (FUNARG) LABEL.

```
((EQ(CAR FORM)(QUOTE QUOTE))(CAR(CDR FORM)))                                    LISP 131
((EQ(CAR FORM)(QUOTE FUNCTION))(LIST(QUOTE FUNARG)(CAR(CDR FORM))              LISP 132
      ALIST))                                                                   LISP 133
((EQ(CAR FORM)(QUOTE COND))(EVCONS(CDR FORM) ALIST))                            LISP 134
((EQ(CAR FORM)(QUOTE PROG))(PROGS(CDR FORM) ALIST ))                            LISP 135
((ATOM(CAR FORM))                                                               LISP 136
      (COND                                                                     LISP 137
      ((GET(CAR FORM)(QUOTE EXPR))(APPLY*(GET(CAR FORM)(QUOTE EXPR))LISP 138
           (EVLIS(CDR FORM) ALIST) ALIST))                                      LISP 139
      ((GET(CAR FORM)(QUOTE FEXPR))(APPLY*(GET(CAR FORM)(QUOTE FEXPRLISP 140
           ))(LIST(CDR FORM) ALIST) ALIST))                                     LISP 141
      ((GET(CAR FORM)(QUOTE SUBR))                                              LISP 142
           (COND                                                                LISP 143
           ((EQ(CAR FORM)(QUOTE CAR))(CAR(CAR(EVLIS(CDR FORM)                   LISP 144
           ALIST))))                                                            LISP 145
           ((EQ(CAR FORM)(QUOTE CDR))(CDR(CAR(EVLIS(CDR FORM)                   LISP 146
           ALIST))))                                                            LISP 147
           ((EQ(CAR FORM)(QUOTE CONS))(CONS(CAR(EVLIS(LIST                      LISP 148
           (CAR(CDR FORM)))ALIST))(CAR(EVLIS(CDR(CDR FORM))                     LISP 149
           ALIST))))                                                            LISP 150
           ((EQ(CAR FORM)(QUOTE ATOM))(ATOM(CAR(EVLIS(CDR FORM)                 LISP 151
           ALIST))))                                                            LISP 152
           ((EQ(CAR FORM)(QUOTE EQ))(EQ(CAR(EVLIS(LIST                         LISP 153
           (CAR(CDR FORM)))ALIST))(CAR(EVLIS(CDR(CDR FORM))                     LISP 154
           ALIST))))                                                            LISP 155
           ((EQ(CAR FORM)(QUOTE ERROR))(ERROR(CAR(EVLIS                         LISP 156
           (CDR FORM) ALIST))))                                                 LISP 157
           ((EQ(CAR FORM)(QUOTE NUMBERP))(NUMBERP(CAR(EVLIS                     LISP 158
           (CDR FORM) ALIST))))                                                 LISP 159
           ((EQ(CAR FORM)(QUOTE GET))(GET(CAR(EVLIS(LIST                        LISP 160
           (CAR(CDR FORM)))A_IST))(CAR(EVLIS(CDR(CDR FORM))                     LISP 161
           ALIST))))                                                            LISP 162
           ((EQ(CAR FORM)(QUOTE NCONC ))(NCONC*(CAR(EVLIS(LIST                  LISP 163
           (CAR(CDR FORM)))A_IST))(CAR(EVLIS(CDR(CDR FORM))                     LISP 164
           ALIST))))                                                            LISP 165
           ((EQ(CAR FORM)(QUOTE MAPLIST ))(MAPLIST*(CAR(EVLIS(LIST LISP 166
           (CAR(CDR FORM)))A_IST))(CAR(EVLIS(CDR(CDR FORM))                     LISP 167
           ALIST))))                                                            LISP 168
           ((EQ(CAR FORM)(QUOTE EQUAL ))(EQUAL*(CAR(EVLIS(LIST                 LISP 169
           (CAR(CDR FORM)))A_IST))(CAR(EVLIS(CDR(CDR FORM))                     LISP 170
           ALIST))))                                                            LISP 171
           ((EQ(CAR FORM)(QUOTE RETURN))FORM)                                   LISP 172
           ((EQ(CAR FORM)(QUOTE NULL ))(NULL*(CAR(EVLIS(CDR FORM)               LISP 173
           ALIST))))                                                            LISP 174
      ((GET(CAR FORM)(QUOTE FSUBR))                                             LISP 175
           (COND                                                                LISP 176
           ((EQ(CAR FORM)(QUOTE LIST))(EVLIS*(CDR FORM)ALIST))                  LISP 177
           ((EQ(CAR FORM)(QUOTE SETQ))FORM)                                     LISP 178
           ((EQ(CAR FORM)(QUOTE GO))FORM)                                       LISP 179
           ((EQ(CAR FORM)(QUOTE OR))(OR*(CDR FORM))))                           LISP 180
      (T(EVAL*(CONS(CDR(ASSOC*(CAR FORM)A_IST(QUOTE(LAMBDA)))(ERROR LISP 181
           (QUOTE A9))))))(CDR FORM)) ALIST))                                   LISP 182
 (T(APPLY*(CAR FORM)(EVLIS*(CDR FORM)ALIST) ALIST))))                           LISP 183
?*************************************************************************?LISP 184
?                                                                       ?LISP 185
?                          E V C O N S                                  ?LISP 186
?                                                                       ?LISP 187
?-----------------------------------------------------------------------?LISP 188
?     EVCON* EVALUATES PROPOSITIONAL TERMS, IN ORDER, AND RETURNS       ?LISP 189
?   THE FORM FOLLOWING THE FIRST TRUE PREDICATE.                        ?LISP 190
?                                                                       ?LISP 191
?*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-?LISP 192
(EVCON*(LAMBDA(CONDITION ALIST)                                                 LISP 193
   (COND                                                                        LISP 194
   ((NULL* CONDITION)(ERROR(QUOTE A3)))                                         LISP 195
   ((EVAL*(CAR(CAR CONDITION))A_IST)(EVAL*(CAR(CDR(CAR CONDITION)))             LISP 196
```

```
          ALIST))                                                              LISP 197
    (T(EVCON$(CDR CONDITION) A_IST))    )     ))                             LISP 198
?*****************************************************************************?LISP 199
?                                                                              ?LISP 200
?                              E V L I S $                                     ?LISP 201
?                                                                              ?LISP 202
?------------------------------------------------------------------------------?LISP 203
?      EVLIS$ TAKES AS ITS ARGUMENTS A LIST OF FORMS TO BE                     ?LISP 204
?    EVALUATED AND THE ASSOCIATION _IST.  EVLIS$ EVALUATES EACH OF THE         ?LISP 205
?    FORMS AND RETURNS A LIST OF THE EVALUATED FORMS.                          ?LISP 206
?*****************************************************************************?LISP 207
 (EVLIS$(LAMBDA(ELIST ALIST)                                                   LISP 208
    (MAPLIST$ ELIST (QUOTE(LAMBDA(ARG)(EVAL$(CAR ARG) ALIST))))      ))       LISP 209
?*****************************************************************************?LISP 210
?                                                                              ?LISP 211
?                              S A S S O C $                                   ?LISP 212
?                                                                              ?LISP 213
?------------------------------------------------------------------------------?LISP 214
?      SASSOC$ SEARCHES THE ASSOCIATION LIST FOR AN ATOMIC SYMBOL              ?LISP 215
?    AND RETURNS THE BOUND PAIR.                                               ?LISP 216
?*****************************************************************************?LISP 217
 (SASSOC$(LAMBDA(FX ALIST ERRFUN)                                              LISP 218
    (COND                                                                      LISP 219
    ((NULL$ ALIST)(ERRFUN))                                                    LISP 220
    ((EQ(CAR(CAR ALIST)) FX)(CAR A_IST))                                       LISP 221
    (T(SASSOC$ FX (CDR ALIST) ERRFUN))    )     ))                           LISP 222
?*****************************************************************************?LISP 223
?                                                                              ?LISP 224
?                                P A I R $                                     ?LISP 225
?                                                                              ?LISP 226
?------------------------------------------------------------------------------?LISP 227
?      PAIR$ TAKES AS ITS ARGUMENTS TWO LISTS OF EQUAL LENGTH.                 ?LISP 228
?    PAIR$ BINDS CORRESPONDING ELEMENTS OF EACH LIST AND ADDS THEM             ?LISP 229
?    TO THE FRONT OF THE ASSOCIATION _IST.                                     ?LISP 230
?*****************************************************************************?LISP 231
 (PAIR$(LAMBDA(ARG1 ARG2)                                                      LISP 232
    (PROG(A1 A2 PLIST)                                                         LISP 233
         (SETQ A1 ARG1)                                                        LISP 234
         (SETQ A2 ARG2)                                                        LISP 235
    AA   (COND                                                                 LISP 236
         ((NULL$ A1)(COND                                                      LISP 237
                   ((NULL$ A2)(RETURN PLIST))                                  LISP 238
                   (T(ERROR(QUOTE F2)))  )  )                                  LISP 239
         ((NULL$ A2)(ERROR(QUOTE F3)))       )                                 LISP 240
         (SETQ PLIST (CONS(CONS(CAR A1)(CAR A2)) PLIST))                       LISP 241
         (SETQ A1 (CDR A1))                                                    LISP 242
         (SETQ A2 (CDR A2))                                                    LISP 243
         (GO AA)  )      ))                                                    LISP 244
 (MAPLIST$(LAMBDA(ELIST FUNC)                                                  LISP 245
    (COND                                                                      LISP 246
    ((NULL$ ELIST ) NIL )                                                      LISP 247
    (T(CONS(FUNC ELIST)(MAPLIST$(CDR ELIST) FUNC)))    )     ))                LISP 248
 (NCONC$(LAMBDA(ARG1 ARG2)                                                     LISP 249
    (PROG(CLIST)                                                               LISP 250
         (COND                                                                 LISP 251
         ((NULL$ ARG1)(RETURN ARG2))                                           LISP 252
         (T(SETQ CLIST ARG1))      )                                           LISP 253
    A    (COND                                                                 LISP 254
         ((NULL$(CDR CLIST))(GO B))                                            LISP 255
         (T(SETQ CLIST (CDR CLIST)))      )                                    LISP 256
         (GO A)                                                                LISP 257
    B    (RPLACD CLIST ARG2)                                                   LISP 258
         (RETURN ARG1)      )          ))                                     LISP 259
 (EQUAL$(LAMBDA(ARG1 ARG2)                                                     LISP 260
    (COND                                                                      LISP 261
    ((ATOM ARG1)(COND                                                          LISP 262
```

```
                ((ATOM ARG2)(EQ ARG1 ARG2))                                    LISP 263
                (T  NIL)      ))                                               LISP 264
    ((EQUAL$(CAR ARG1)(CAR ARG2))(EQUAL$(CDR ARG1)(CDR ARG2)))                 LISP 265
    ( T  NIL)   )     ))                                                       LISP 266
 (NULL$(LAMBDA(ARG)                                                            LISP 267
    (COND                                                                      LISP 268
    ((ATOM ARG)(EQ ARG NIL))                                                   LISP 269
    ( T  NIL)   )          ))                                                  LISP 270
 (DEFINE$(LAMBDA(ARGLIST)                                                      LISP 271
   (PROG(AL NL)                                                                LISP 272
     (SETQ AL ARGLIST)                                                         LISP 273
 TOP (COND                                                                     LISP 274
       ((NULL$ AL)(RETURN NL))                                                 LISP 275
       ((PUT(CAR(CAR AL))(QUOTE EXPR)(CAR(CDR(CAR AL))))  )                   LISP 276
         (SETQ NL(CONS(CAR(CAR AL)) NL ))))                                    LISP 277
       (COND                                                                   LISP 278
         ((NULL$ AL)(RETURN NL))                                               LISP 279
         ((T(SETQ AL(CDR AL))))   )                                            LISP 280
       (GO TOP)   )   ))                                                       LISP 281
(PROG$(LAMBDA(PBODY ALIST)                                                     LISP 282
 (PROG(B A GLIST BCAR TEMP)                                                    LISP 283
    (SETQ B PBODY)                                                             LISP 284
    (SETQ A(NCONC$(PAIR$(CAR B)(NLIST$(LENGTH$(CAR B))))) ALIST ))             LISP 285
    (SETQ GLIST (GOLIST$(CDR B)))                                              LISP 286
 L1   (SETQ B(CDR B))                                                          LISP 287
 L2   (SETQ BCAR(CAR B))                                                       LISP 288
 L3   (COND                                                                    LISP 289
        ((NULL$ BCAR)(ERROR(QUOTE A3)))                                        LISP 290
        ((ATOM BCAR)(GO L1))                                                   LISP 291
        ((EQ(CAR BCAR)(QUOTE SETQ))                                            LISP 292
          (SETQ A(REPLACE$(CAR(CDR BCAR))                                      LISP 293
          (EVAL$(CAR(CDR(CDR BCAR)))A)A))                                      LISP 294
          (GO L1) )                                                            LISP 295
        ((EQ(CAR BCAR)(QUOTE GO))                                              LISP 296
          (SETQ B(CDR(SASSOC$(CAR(CDR BCAR)) GLIST                             LISP 297
          (QUOTE(LAMBDA)((ERROR(QUOTE A6)))))))                                LISP 298
          (GO L2) )                                                            LISP 299
        ((EQ(CAR BCAR)(QUOTE RETURN))                                          LISP 300
          (RETURN(EVAL$(CAR(CDR BCAR))A)) )                                    LISP 301
        (T(SETQ TEMP(EVAL$ BCAR A))                                            LISP 302
          (COND                                                                LISP 303
          ((NOT(ATOM TEMP))                                                    LISP 304
             (COND                                                             LISP 305
             ((OR$(EQ(CAR TEMP)(QUOTE SETQ))                                   LISP 306
                  ((EQ(CAR TEMP)(QUOTE GO))                                    LISP 307
                  ((EQ(CAR TEMP)(QUOTE RETURN)) )                              LISP 308
                    (SETQ BCAR TEMP)(GO L3) )                                  LISP 309
             (T(GO L1)))) )                                                    LISP 310
          (T(GO L1)))))) ))                                                    LISP 311
(REPLACE$(LAMBDA(SUB OBJ ALIST)                                                LISP 312
    (COND                                                                      LISP 313
     ((NULL$ A_IST)(ERROR(QUOTE A8)))                                          LISP 314
     ((EQ(CAAR ALIST) SUB )(CONS(CONS SUB OBJ)(CDR ALIST)))                    LISP 315
     (T(CONS(CAR ALIST)(REPLACE$ SUB OBJ(CDR A_IST))))   )   ))                LISP 316
(NLIST$(LAMBDA(NUM)                                                            LISP 317
    (COND                                                                      LISP 318
     ((ZEROP NUM) NIL )                                                        LISP 319
     (T(CONS NIL (NLIST$ (SUB1 NUM))))   )   ))                                LISP 320
     (GOLIST$(LAMBDA(GL)                                                       LISP 321
           (COND                                                               LISP 322
           ((NULL$ GL) NIL )                                                   LISP 323
           ((ATOM(CAR GL))(NCONC$(PAIR$(LIST(CAR GL))(LIST(CDR GL))))         LISP 324
                 (GO_IST$(CDR G_)))                                            LISP 325
           (T(GO_IST$(CDR GL)))   )   ))                                       LISP 326
 (OR$(LAMBDA(ORL ALIST)                                                        LISP 327
       (COND                                                                   LISP 328
```

```
        ((NULL$ ORL)NIL)                                                LISP 329
        ((EVAL$(CAR ORL)ALIST) T )                                      LISP 330
       (T(OR$(CDR ORL)ALIST)))    )    ))                             LISP 331
(LENGTH$(LAMBDA(SEXP)                                                   LISP 332
 (PROG(LEN)                                                             LISP 333
  (SETQ LEN 0)                                                          LISP 334
 AA (COND                                                               LISP 335
  ((NULL$ SEXP)(RETURN LEN))                                            LISP 336
  (T(SETQ SEXP(CDR SEXP)))    )                                         LISP 337
  (SETQ LEN(ADD1 LEN))                                                  LISP 338
  (GO AA)    )    ))                                                    LISP 339
)))                                                                     LISP 340
```