

The Standard Lisp Report

Jed Marti
A. C. Hearn
M. L. Griss
C. Griss

1 Introduction

Although the programming language LISP was first formulated in 1960 [7], a widely accepted standard has never appeared. As a result, various dialects of LISP were produced [1, 2, 6, 10, 8, 9] in some cases several on the same machine! Consequently, a user often faces considerable difficulty in moving programs from one system to another. In addition, it is difficult to write and use programs which depend on the structure of the source code such as translators, editors and cross-reference programs.

In 1969, a model for such a standard was produced [4] as part of a general effort to make a large LISP based algebraic manipulation program, REDUCE [5], as portable as possible. The goal of this work was to define a uniform subset of LISP 1.5 and its variants so that programs written in this subset could run on any reasonable LISP system.

In the intervening years, two deficiencies in the approach taken in Ref. [4] have emerged. First in order to be as general as possible, the specific semantics and values of several key functions were left undefined. Consequently, programs built on this subset could not make any assumptions about the form of the values of such functions. The second deficiency related to the proposed method of implementation of this language. The model considered in effect two versions of LISP on any given machine, namely Standard LISP and the LISP of the host machine (which we shall refer to as Target LISP). This meant that if any definition was stored in interpretive form, it would vary from implementation to implementation, and consequently one could not write programs in Standard LISP which needed to assume any knowledge about the structure of such forms. This deficiency became apparent during recent work on the development of a portable compiler for LISP [3]. Clearly a compiler has to know precisely the structure of its source code; we concluded that the appropriate source was Standard LISP and not Target LISP.

With these thoughts in mind we decided to attempt again a definition of Standard LISP. However, our approach this time is more aggressive. In this document we define a standard for a reasonably large subset of LISP with as precise

as possible a statement about the semantics of each function. Secondly, we now require that the target machine interpreter be modified or written to support this standard, rather than mapping Standard LISP onto Target LISP as previously.

We have spent countless hours in discussion over many of the definitions given in this report. We have also drawn on the help and advice of a lot of friends whose names are given in the Acknowledgements. Wherever possible, we have used the definition of a function as given in the LISP 1.5 Programmer's Manual [7] and have only deviated where we felt it desirable in the light of LISP programming experience since that time. In particular, we have given considerable thought to the question of variable bindings and the definition of the evaluator functions EVAL and APPLY. We have also abandoned the previous definition of LISP arrays in favor of the more accepted idea of a vector which most modern LISP systems support. These are the places where we have strayed furthest from the conventional definitions, but we feel that the consistency which results from our approach is worth the redefinition.

We have avoided entirely in this report problems which arise from environment passing, such as those represented by the FUNARG problem. We do not necessarily exclude these considerations from our standard, but in this report have decided to avoid the controversy which they create. The semantic differences between compiled and interpreted functions is the topic of another paper [3]. Only functions which affect the compiler in a general way make reference to it.

This document is not intended as an introduction to LISP rather it is assumed that the reader is already familiar with some version. The document is thus intended as an arbiter of the syntax and semantics of Standard LISP. However, since it is not intended as an implementation description, we deliberately leave unspecified many of the details on which an actual implementation depends. For example, while we assume the existence of a symbol table for atoms (the "object list" in LISP terminology), we do not specify its structure, since conventional LISP programming does not require this information. Our ultimate goal, however, is to remedy this by defining an interpreter for Standard LISP which is sufficiently complete that its implementation on any given computer will be straightforward and precise. At that time, we shall produce an implementation level specification for Standard LISP which will extend the description of the primitive functions defined herein by introducing a new set of lower level primitive functions in which the structure of the symbol table, heap and so on may be defined.

The plan of this chapter is as follows. In Section 2 we describe the various data types used in Standard LISP. In Section 3, a description of all Standard LISP functions is presented, organized by type. These functions are defined in an RLISP syntax which is easier to read than LISP S-expressions. Section 4 describes global variables which control the operation of Standard LISP.

2 Preliminaries

2.1 Primitive Data Types

integer Integers are also called "fixed" numbers. The magnitude of an integer is unrestricted. Integers in the LISP input stream are recognized by the grammar:

```

<digit> ::= 0|1|2|3|4|5|6|7|8|9
<unsigned-integer> ::= <digit>|<unsigned-integer><digit>
<integer> ::= <unsigned-integer> |
              +<unsigned-integer> |
              -<unsigned-integer>

```

floating - Any floating point number. The precision of floating point numbers is determined solely by the implementation. In BNF floating point numbers are recognized by the grammar:

```

<base> ::= <unsigned-integer>.|.<unsigned-integer>|
          <unsigned-integer>.<unsigned-integer>
<unsigned-floating> ::= <base>|
          <base>E<unsigned-integer>|
          <base>E-<unsigned-integer>|
          <base>E+<unsigned-integer>
<floating> ::= <unsigned-floating>|
              +<unsigned-floating>|-<unsigned-floating>

```

id An identifier is a string of characters which may have the following items associated with it.

print name The characters of the identifier.

flags An identifier may be tagged with a flag. Access is by the FLAG, REMFLAG, and FLAGP functions defined in section 3.4 on page 13.

properties An identifier may have an indicator-value pair associated with it. Access is by the PUT, GET, and REMPROP functions defined in section 3.4 on page 13.

values/functions An identifier may have a value associated with it. Access to values is by SET and SETQ defined in section 3.6 on page 16. The method by which the value is attached to the identifier is known as the binding type, being one of LOCAL, GLOBAL, or FLUID. Access to the binding type is by the GLOBAL, GLOBALP, FLUID, FLUIDP, and UNFLUID functions.

An identifier may have a function or macro associated with it. Access is by the PUTD, GETD, and REMD functions (see "Function Definition", section 3.5, on page 14). An identifier may not have both a function and a value associated with it.

OBLIST entry An identifier may be entered and removed from a structure called the OBLIST. Its presence on the OBLIST does not directly affect the other properties. Access to the OBLIST is by the INTERN, REMOB, and READ functions.

The maximum length of a Standard LISP identifier is 24 characters (excluding occurrences of the escape character !) but an implementation may allow more. Special characters (digits in the first position and punctuation) must be prefixed with an escape character, an ! in Standard LISP. In BNF identifiers are recognized by the grammar:

```

<special-character> ::= !<any-character>
<alphabetic> ::=
    A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|
    a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<lead-character> ::= <special-character>|<alphabetic>
<regular-character> ::= <lead-character>|<digit>
<last-part> ::= <regular-character> |
                <last-part><regular-character>
<id> ::= <lead-character>|<lead-character><last-part>

```

Note: Using lower case letters in identifiers may cause portability problems. Lower case letters are automatically converted to upper case when the !*RAISE flag is T.

string A set of characters enclosed in double quotes as in "THIS IS A STRING". A quote is included by doubling it as in "HE SAID, ""LISP""". The maximum size of strings is 80 characters but an implementation may allow more. Strings are not part of the OBLIST and are considered constants like numbers, vectors, and function-pointers.

dotted-pair A primitive structure which has a left and right part. A notation called *dot-notation* is used for dotted pairs and takes the form:

```
(<left-part> . <right-part>)
```

The *<left-part>* is known as the CAR portion and the *<right-part>* as the CDR portion. The left and right parts may be of any type. Spaces are used to resolve ambiguity with floating point numbers.

vector A primitive uniform structure in which an integer index is used to access random values in the structure. The individual elements of a vector may be of any type. Access to vectors is restricted to functions defined in "Vectors" section 3.9 on page 21. A notation for vectors, *vector-notation*, has the elements of a vector surrounded by square brackets¹

```

<elements> ::= <any>|<any> <elements>
<vector> ::= [<elements>]

```

function-pointer An implementation may have functions which deal with specific data types other than those listed. The use of these entities is to be avoided with the exception of a restricted use of the function-pointer, an access method to compiled EXPRs and FEXPRs. A particular function-pointer must remain valid throughout execution. Systems which change the location of a function must use either an indirect reference or change all occurrences of the associated value. There are two classes of use of

¹Vector elements are not separated by commas as in the published version of this document.

function-pointers, those which are supported by Standard LISP but are not well defined, and those which are well defined.

Not well defined Function pointers may be displayed by the print functions or expanded by EXPLODE. The value appears in the convention of the implementation site. The value is not defined in Standard LISP. Function pointers may be created by COMPRESS in the format used for printing but the value used is not defined in Standard LISP. Function pointers may be created by functions which deal with compiled function loading. Again, the values created are not well defined in Standard LISP.

Well defined The function pointer associated with an EXPR or FEXPR may be retrieved by GETD and is valid as long as Standard LISP is in execution. Function pointers may be stored using PUTD, PUT, SETQ and the like or by being bound to variables. Function pointers may be checked for equivalence by EQ. The value may be checked for being a function pointer by the CODEP function.

2.2 Classes of Primitive Data Types

The classes of primitive types are a notational convenience for describing the properties of functions.

boolean The set of global variables {T,NIL}, or their respective values, {T, NIL}.

extra-boolean Any value in the system. Anything that is not NIL has the boolean interpretation T.

ftype The class of definable function types. The set of ids {EXPR, FEXPR, MACRO}.

number The set of {integer, floating}.

constant The set of {integer, floating, string, vector, function-pointer}. Constants evaluate to themselves (see the definition of EVAL in “The Interpreter”, section 3.14 on page 32).

any The set of {integer, floating, string, id, dotted-pair, vector, function-pointer}. An S-expression is another term for any. All Standard LISP entities have some value unless an ERROR occurs during evaluation or the function causes transfer of control (such as GO and RETURN).

atom The set {any}-{dotted-pair}.

2.3 Structures

Structures are entities created out of the primitive types by the use of dotted-pairs. Lists are structures very commonly required as actual parameters to functions. Where a list of homogeneous entities is required by a function this

class will be denoted by $\langle \mathbf{xxx-list} \rangle$ where \mathbf{xxx} is the name of a class of primitives or structures. Thus a list of ids is an *id-list*, a list of integers an *integer-list* and so on.

list A list is recursively defined as NIL or the dotted-pair (any . list). A special notation called *list-notation* is used to represent lists. List-notation eliminates extra parentheses and dots. The list (a . (b . (c . NIL))) in list notation is (a b c). List-notation and dot-notation may be mixed as in (a b . c) or (a (b . c) d) which are (a . (b . c)) and (a . ((b . c) . (d . NIL))). In BNF lists are recognized by the grammar:

$$\begin{aligned} \langle \textit{left-part} \rangle &::= (\mid \langle \textit{left-part} \rangle \langle \textit{any} \rangle \\ \langle \textit{list} \rangle &::= \langle \textit{left-part} \rangle \mid \langle \textit{left-part} \rangle . \langle \textit{any} \rangle \end{aligned}$$

Note: () is an alternate input representation of NIL.

alist An association list; each element of the list is a dotted-pair, the CAR part being a key associated with the value in the CDR part.

cond-form A cond-form is a list of 2 element lists of the form:

$$(\mathbf{ANTECEDENT:} \mathit{any} \mathbf{CONSEQUENT:} \mathit{any})$$

The first element will henceforth be known as the antecedent and the second as the consequent. The antecedent must have a value. The consequent may have a value or an occurrence of GO or RETURN as described in the “Program Feature Functions”, section 3.7 on page 18.

lambda A LAMBDA expression which must have the form (in list notation): (LAMBDA parameters body). “parameters” is a list of formal parameters for “body” an S-expression to be evaluated. The semantics of the evaluation are defined with the EVAL function (see “The Interpreter”, section 3.14 on page 32).

function A LAMBDA expression or a function-pointer to a function. A function is always evaluated as an EVAL, SPREAD form.

2.4 Function Descriptions

Each function is provided with a prototypical header line. Each formal parameter is given a name and suffixed with its allowed type. Lower case, italic tokens are names of classes and upper case, bold face, tokens are parameter names referred to in the definition. The type of the value returned by the function (if any) is suffixed to the parameter list. If it is not commonly used the parameter type may be a specific set enclosed in brackets {...}. For example:

$$\text{PUTD}(\mathbf{FNAME:} \mathit{id}, \mathbf{TYPE:} \mathit{ftype}, \mathbf{BODY:} \{ \mathit{lambda}, \mathit{function-pointer} \}) : \mathit{id}$$

PUTD is a function with three parameters. The parameter FNAME is an id to be the name of the function being defined. TYPE is the type of the function being defined and BODY is a lambda expression or a function-pointer. PUTD returns the name of the function being defined.

Functions which accept formal parameter lists of arbitrary length have the type class and parameter enclosed in square brackets indicating that zero or more occurrences of that argument are permitted. For example:

`AND([U:any]):extra-boolean`

AND is a function which accepts zero or more arguments which may be of any type.

2.5 Function Types

EVAL type functions are those which are invoked with evaluated arguments. NOEVAL functions are invoked with unevaluated arguments. SPREAD type functions have their arguments passed in one-to-one correspondence with their formal parameters. NOSPREAD functions receive their arguments as a single list. EVAL, SPREAD functions are associated with EXPRs and NOEVAL, NOSPREAD functions with FEXPRs. EVAL, NOSPREAD and NOEVAL, SPREAD functions can be simulated using NOEVAL, NOSPREAD functions or MACROS.

EVAL, SPREAD type functions may have a maximum of 15 parameters. There is no limit on the number of parameters a NOEVAL, NOSPREAD function or MACRO may have.

In the context of the description of an EVAL, SPREAD function, then we speak of the formal parameters we mean their actual values. However, in a NOEVAL, NOSPREAD function it is the unevaluated actual parameters.

A third function type, the MACRO, implements functions which create S-expressions based on actual parameters. When a macro invocation is encountered, the body of the macro, a lambda expression, is invoked as a NOEVAL, NOSPREAD function with the macro's invocation bound as a list to the macros single formal parameter. When the macro has been evaluated the resulting S-expression is reevaluated. The description of the EVAL and EXPAND functions provide precise details.

2.6 Error and Warning Messages

Many functions detect errors. The description of such functions will include these error conditions and suggested formats for display of the generated error messages. A call on the ERROR function is implied but the error number is not specified by Standard LISP. In some cases a warning message is sufficient. To distinguish between errors and warnings, errors are prefixed with five asterisks and warnings with only three.

Primitive functions check arguments that must be of a certain primitive type for being of that type and display an error message if the argument is not correct. The type mismatch error always takes the form:

***** PARAMETER not TYPE for FN

Here PARAMETER is the unacceptable actual parameter, TYPE is the type that PARAMETER was supposed to be. FN is the name of the function that detected the error.

2.7 Comments

The character % signals the start of a comment, text to be ignored during parsing. A comment is terminated by the end of the line it is on. The function READCH must be able to read a comment one character at a time. Comments are transparent to the function READ. % may occur as a character in identifiers by preceding it with the escape character !.

3 Functions

3.1 Elementary Predicates

Functions in this section return T when the condition defined is met and NIL when it is not. Defined are type checking functions and elementary comparisons.

ATOM(**U**:any):boolean *eval, spread*

Returns T if U is not a pair.

```
EXPR PROCEDURE ATOM(U);
  NULL PAIRP U;
```

CODEP(**U**:any):boolean *eval, spread*

Returns T if U is a function-pointer.

CONSTANTP(**U**:any):boolean *eval, spread*

Returns T if U is a constant (a number, string, function-pointer, or vector).

```
EXPR PROCEDURE CONSTANTP(U);
  NULL OR(PAIRP U, IDP U);
```

EQ(**U**:any, **V**:any):boolean *eval, spread*

Returns T if U points to the same object as V. EQ is not a reliable comparison between numeric arguments.

EQN(*U: any, V: any*): *boolean* *eval, spread*
 Returns T if U and V are EQ or if U and V are numbers and have the same value and type.

EQUAL(*U: any, V: any*): *boolean* *eval, spread*
 Returns T if U and V are the same. Dotted-pairs are compared recursively to the bottom levels of their trees. Vectors must have identical dimensions and EQUAL values in all positions. Strings must have identical characters. Function pointers must have EQ values. Other atoms must be EQN equal.

FIXP(*U: any*): *boolean* *eval, spread*
 Returns T if U is an integer (a fixed number).

FLOATP(*U: any*): *boolean* *eval, spread*
 Returns T if U is a floating point number.

IDP(*U: any*): *boolean* *eval, spread*
 Returns T if U is an id.

MINUSP(*U: any*): *boolean* *eval, spread*
 Returns T if U is a number and less than 0. If U is not a number or is a positive number, NIL is returned.
 EXPR PROCEDURE MINUSP(U);
 IF NUMBERP U THEN LESSP(U, 0) ELSE NIL;

NULL(*U: any*): *boolean* *eval, spread*
 Returns T if U is NIL.
 EXPR PROCEDURE NULL(U);
 U EQ NIL;

NUMBERP(*U: any*): *boolean* *eval, spread*
 Returns T if U is a number (integer or floating).
 EXPR PROCEDURE NUMBERP(U);
 IF OR(FIXP U, FLOATP U) THEN T ELSE NIL;

ONEP(*U: any*): *boolean* *eval, spread.*

Returns T if U is a number and has the value 1 or 1.0. Returns NIL otherwise.^a

```
EXPR PROCEDURE ONEP(U);
  OR(EQN(U, 1), EQN(U, 1.0));
```

^aThe definition in the published report is incorrect as it does not return T for U of 1.0.

PAIRP(*U: any*): *boolean* *eval, spread*

Returns T if U is a dotted-pair.

STRINGP(*U: any*): *boolean* *eval, spread*

Returns T if U is a string.

VECTORP(*U: any*): *boolean* *eval, spread*

Returns T if U is a vector.

ZEROP(*U: any*): *boolean* *eval, spread.*

Returns T if U is a number and has the value 0 or 0.0. Returns NIL otherwise.^a

```
EXPR PROCEDURE ZEROP(U);
  OR(EQN(U, 0), EQN(U, 0.0));
```

^aThe definition in the published report is incorrect as it does not return T for U of 0.0.

3.2 Functions on Dotted-Pairs

The following are elementary functions on dotted-pairs. All functions in this section which require dotted-pairs as parameters detect a type mismatch error if the actual parameter is not a dotted-pair.

CAR(*U: dotted-pair*): *any* *eval, spread*

CAR(CONS(a, b)) → a. The left part of U is returned. The type mismatch error occurs if U is not a dotted-pair.

CDR(*U: dotted-pair*): *any* *eval, spread*

CDR(CONS(a, b)) → b. The right part of U is returned. The type mismatch error occurs if U is not a dotted-pair.

The composites of CAR and CDR are supported up to 4 levels, namely:

```

CAAAAR  CAAAR  CAAR
CAAADR  CAADR  CADR
CAADAR  CADAR  CDAR
CAADDR  CADDR  CDDR
CADAAR  CDAAR
CADADR  CDADR
CADDAR  CDDAR
CADDRR  CDDDR
CDAAAR
CDAADR
CDADAR
CDADDR
CDDAAR
CDDADR
CDDDRR
CDDDDR

```

CONS(*U: any, V: any*):*dotted-pair* *eval, spread*

Returns a dotted-pair which is not EQ to anything and has U as its CAR part and V as its CDR part.

LIST([*U: any*]):*list* *noeval, nospread, or macro*

A list of the evaluation of each element of U is returned. The order of evaluation need not be first to last as the following definition implies.^a

```

FEXPR PROCEDURE LIST(U);
  EVLIS U;

```

^aThe published report's definition implies a specific ordering.

RPLACA(*U: dotted-pair, V: any*):*dotted-pair* *eval, spread*

The CAR portion of the dotted-pair U is replaced by V. If dotted-pair U is (a . b) then (V . b) is returned. The type mismatch error occurs if U is not a dotted-pair.

RPLACD(*U: dotted-pair, V: any*):*dotted-pair* *eval, spread*

The CDR portion of the dotted-pair U is replaced by V. If dotted-pair U is (a . b) then (a . V) is returned. The type mismatch error occurs if U is not a dotted-pair.

3.3 Identifiers

The following functions deal with identifiers and the OBLIST, the structure of which is not defined. The function of the OBLIST is to provide a symbol table for identifiers created during input. Identifiers created by READ which have the same characters will therefore refer to the same object (see the EQ function in “Elementary Predicates”, section 3.1 on page 8).

COMPRESS(U: *id-list*):{ *atom-vector*} *eval, spread*

U is a list of single character identifiers which is built into a Standard LISP entity and returned. Recognized are numbers, strings, and identifiers with the escape character prefixing special characters. The formats of these items appear in “Primitive Data Types” section 2.1 on page 3. Identifiers are not interned on the OBLIST. Function pointers may be compressed but this is an undefined use. If an entity cannot be parsed out of U or characters are left over after parsing an error occurs:

***** Poorly formed atom in COMPRESS

EXPLODE(U:{ *atom*}-{ *vector* }):*id-list* *eval, spread*

Returned is a list of interned characters representing the characters to print of the value of U. The primitive data types have these formats:
integer Leading zeroes are suppressed and a minus sign prefixes the digits if the integer is negative.

floating The value appears in the format [-]0.nn...nnE[-]mm if the magnitude of the number is too large or small to display in [-]nnnn.nnnn format. The crossover point is determined by the implementation.

id The characters of the print name of the identifier are produced with special characters prefixed with the escape character.

string The characters of the string are produced surrounded by double quotes "...".

function-pointer The value of the function-pointer is created as a list of characters conforming to the conventions of the system site.

The type mismatch error occurs if U is not a number, identifier, string, or function-pointer.

GENSYM():*identifier* *eval, spread*

Creates an identifier which is not interned on the OBLIST and consequently not EQ to anything else.

INTERN(U:{id,string}):id *eval, spread*

INTERN searches the OBLIST for an identifier with the same print name as U and returns the identifier on the OBLIST if a match is found. Any properties and global values associated with U may be lost. If U does not match any entry, a new one is created and returned. If U has more than the maximum number of characters permitted by the implementation (the minimum number is 24) an error occurs:

***** Too many characters to INTERN

REMOB(U:id):id *eval, spread*

If U is present on the OBLIST it is removed. This does not affect U having properties, flags, functions and the like. U is returned.

3.4 Property List Functions

With each id in the system is a “property list”, a set of entities which are associated with the id for fast access. These entities are called “flags” if their use gives the id a single valued property, and “properties” if the id is to have a multivalued attribute: an indicator with a property.

Flags and indicators may clash, consequently care should be taken to avoid this occurrence. Flagging X with an id which already is an indicator for X may result in that indicator and associated property being lost. Likewise, adding an indicator which is the same id as a flag may result in the flag being destroyed.

FLAG(U:id-list, V:id):NIL *eval, spread*

U is a list of ids which are flagged with V. The effect of FLAG is that FLAGP will have the value T for those ids of U which were flagged. Both V and all the elements of U must be identifiers or the type mismatch error occurs.

FLAGP(U:any, V:any):boolean *eval, spread*

Returns T if U has been previously flagged with V, else NIL. Returns NIL if either U or V is not an id.

GET(*U: any, IND: any*):*any* *eval, spread*

Returns the property associated with indicator IND from the property list of U. If U does not have indicator IND, NIL is returned. GET cannot be used to access functions (use GETD instead).

PUT(*U: id, IND: id, PROP: any*):*any* *eval, spread*

The indicator IND with the property PROP is placed on the property list of the id U. If the action of PUT occurs, the value of PROP is returned. If either of U and IND are not ids the type mismatch error will occur and no property will be placed. PUT cannot be used to define functions (use PUTD instead).

REMFLAG(*U: any-list, V: id*):*NIL* *eval, spread*

Removes the flag V from the property list of each member of the list U. Both V and all the elements of U must be ids or the type mismatch error will occur.

REMPROP(*U: any, IND: any*):*any* *eval, spread*

Removes the property with indicator IND from the property list of U. Returns the removed property or NIL if there was no such indicator.

3.5 Function Definition

Functions in Standard LISP are global entities. To avoid function-variable naming clashes no variable may have the same name as a function.

DE(*FNAME: id, PARAMS: id-list, FN: any*):*id* *noeval, nospread*

The function FN with the formal parameter list PARAMS is added to the set of defined functions with the name FNAME. Any previous definitions of the function are lost. The function created is of type EXPR. If the !*COMP variable is non-NIL, the EXPR is first compiled. The name of the defined function is returned.

```
FEXPR PROCEDURE DE(U);
  PUTD(CAR U, 'EXPR, LIST('LAMBDA, CADR U, CADDR U));
```

DF(FNAME: *id*, PARAM: *id-list*, FN: *any*): *id* *noeval, nospread*

The function FN with formal parameter PARAM is added to the set of defined functions with the name FNAME. Any previous definitions of the function are lost. The function created is of type FEXPR. If the !*COMP variable is T the FEXPR is first compiled. The name of the defined function is returned.

```
FEXPR PROCEDURE DF(U);
  PUTD(CAR U, 'FEXPR, LIST('LAMBDA, CADR U, CADDR U));
```

DM(MNAME: *id*, PARAM: *id-list*, FN: *any*): *id* *noeval, nospread*

The macro FN with the formal parameter PARAM is added to the set of defined functions with the name MNAME. Any previous definitions of the function are overwritten. The function created is of type MACRO. The name of the macro is returned.

```
FEXPR PROCEDURE DM(U);
  PUTD(CAR U, 'MACRO, LIST('LAMBDA, CADR U, CADDR U));
```

GETD(FNAME: *any*): {NIL, *dotted-pair*} *eval, spread*

If FNAME is not the name of a defined function, NIL is returned. If FNAME is a defined function then the dotted-pair

```
(TYPE: ftype . DEF: {function-pointer, lambda})
```

is returned.

PUTD(FNAME: *id*, TYPE: *ftype*, BODY: *function*): *id* *eval, spread*

Creates a function with name FNAME and definition BODY of type TYPE. If PUTD succeeds the name of the defined function is returned. The effect of PUTD is that GETD will return a dotted-pair with the functions type and definition. Likewise the GLOBALP predicate will return T when queried with the function name.

If the function FNAME has already been declared as a GLOBAL or FLUID variable the error:

```
***** FNAME is a non-local variable
```

occurs and the function will not be defined. If function FNAME already exists a warning message will appear:

```
*** FNAME redefined
```

The function defined by PUTD will be compiled before definition if the !*COMP global variable is non-NIL.

REMD(FNAME: *id*):{NIL, dotted-pair} *eval, spread*
 Removes the function named FNAME from the set of defined functions. Returns the (ftype . function) dotted-pair or NIL as does GETD. The global/function attribute of FNAME is removed and the name may be used subsequently as a variable.

3.6 Variables and Bindings

A variable is a place holder for a Standard LISP entity which is said to be bound to the variable. The scope of a variable is the range over which the variable has a defined value. There are three different binding mechanisms in Standard LISP.

Local Binding This type of binding occurs only in compiled functions. Local variables occur as formal parameters in lambda expressions and as PROG form variables. The binding occurs when a lambda expression is evaluated or when a PROG form is executed. The scope of a local variable is the body of the function in which it is defined.

Global Binding Only one binding of a global variable exists at any time allowing direct access to the value bound to the variable. The scope of a global variable is universal. Variables declared GLOBAL may not appear as parameters in lambda expressions or as PROG form variables. A variable must be declared GLOBAL prior to its use as a global variable since the default type for undeclared variables is FLUID.

Fluid Binding Fluid variables are global in scope but may occur as formal parameters or PROG form variables. In interpreted functions all formal parameters and PROG form variables are considered to have fluid binding until changed to local binding by compilation. When fluid variables are used as parameters they are rebound in such a way that the previous binding may be restored. All references to fluid variables are to the currently active binding.

FLUID(IDLIST: *id-list*):NIL *eval, spread*
 The ids in IDLIST are declared as FLUID type variables (ids not previously declared are initialized to NIL). Variables in IDLIST already declared FLUID are ignored. Changing a variable's type from GLOBAL to FLUID is not permissible and results in the error:

***** ID cannot be changed to FLUID

FLUIDP(*U:any*):*boolean* *eval, spread*

If *U* has been declared FLUID (by declaration only) T is returned, otherwise NIL is returned.

GLOBAL(*IDLIST:id-list*):**NIL** *eval, spread*

The ids of *IDLIST* are declared global type variables. If an id has not been declared previously it is initialized to NIL. Variables already declared GLOBAL are ignored. Changing a variables type from FLUID to GLOBAL is not permissible and results in the error:

***** ID cannot be changed to GLOBAL

GLOBALP(*U:any*):*boolean* *eval, spread*

If *U* has been declared GLOBAL or is the name of a defined function, T is returned, else NIL is returned.

SET(*EXP:id*, *VALUE:any*):*any* *eval, spread*

EXP must be an identifier or a type mismatch error occurs. The effect of SET is replacement of the item bound to the identifier by *VALUE*. If the identifier is not a local variable or has not been declared GLOBAL it is automatically declared FLUID with the resulting warning message:

*** EXP declared FLUID

EXP must not evaluate to T or NIL or an error occurs:

***** Cannot change T or NIL

SETQ(VARIABLE: *id*, VALUE: *any*): *any* *noeval, nospread*

If VARIABLE is not local or GLOBAL it is by default declared FLUID and the warning message:

```
*** VARIABLE declared FLUID
```

appears. The value of the current binding of VARIABLE is replaced by the value of VALUE. VARIABLE must not be T or NIL or an error occurs:

```
***** Cannot change T or NIL
```

```
MACRO PROCEDURE SETQ(X);
  LIST('SET, LIST('QUOTE, CADR X), CADDR X);
```

UNFLUID(IDLIST: *id-list*): *NIL* *eval, spread*

The variables in IDLIST that have been declared as FLUID variables are no longer considered as fluid variables. Others are ignored. This affects only compiled functions as free variables in interpreted functions are automatically considered fluid [3].

3.7 Program Feature Functions

These functions provide for explicit control sequencing, and the definition of blocks altering the scope of local variables.

GO(**LABEL**:*id*) *noeval, nospread*

GO alters the normal flow of control within a PROG function. The next statement of a PROG function to be evaluated is immediately preceded by LABEL. A GO may only appear in the following situations:

1. At the top level of a PROG referencing a label which also appears at the top level of the same PROG.
2. As the consequent of a COND item of a COND appearing on the top level of a PROG.
3. As the consequent of a COND item which appears as the consequent of a COND item to any level.
4. As the last statement of a PROGN which appears at the top level of a PROG or in a PROGN appearing in the consequent of a COND to any level subject to the restrictions of 2 and 3.
5. As the last statement of a PROGN within a PROGN or as the consequent of a COND in a PROGN to any level subject to the restrictions of 2, 3 and 4.

If LABEL does not appear at the top level of the PROG in which the GO appears, an error occurs:

```
***** LABEL is not a known label
```

If the GO has been placed in a position not defined by rules 1-5, another error is detected:

```
***** Illegal use of GO to LABEL
```

PROG(**VARS**:*id-list*, [**PROGRAM**:{*id, any*}]):*any* *noeval, nospread*

VARS is a list of ids which are considered fluid when the PROG is interpreted and local when compiled (see “Variables and Bindings”, section 3.6 on page 16). The PROGs variables are allocated space when the PROG form is invoked and are deallocated when the PROG is exited. PROG variables are initialized to NIL. The PROGRAM is a set of expressions to be evaluated in order of their appearance in the PROG function. Identifiers appearing in the top level of the PROGRAM are labels which can be referenced by GO. The value returned by the PROG function is determined by a RETURN function or NIL if the PROG “falls through”.

PROGN([**U**:*any*]):*any* *noeval, nospread*

U is a set of expressions which are executed sequentially. The value returned is the value of the last expression.

PROG2(A:any, B:any) *any* *eval, spread*
 Returns the value of B.
 EXPR PROCEDURE PROG2(A, B);
 B;

RETURN(U:*any*) *eval, spread*
 Within a PROG, RETURN terminates the evaluation of a PROG and returns U as the value of the PROG. The restrictions on the placement of RETURN are exactly those of GO. Improper placement of RETURN results in the error:
 ***** Illegal use of RETURN

3.8 Error Handling

ERROR(NUMBER:*integer*, MESSAGE:*any*) *eval, spread*
 NUMBER and MESSAGE are passed back to a surrounding ERRORSET (the Standard LISP reader has an ERRORSET). MESSAGE is placed in the global variable EMSG!* and the error number becomes the value of the surrounding ERRORSET. FLUID variables and local bindings are unbound to return to the environment of the ERRORSET. Global variables are not affected by the process.

ERRORSET(**U**:*any*, **MSGP**:*boolean*, **TR**:*boolean*):*any* *eval, spread*

If an error occurs during the evaluation of U, the value of NUMBER from the ERROR call is returned as the value of ERRORSET. In addition, if the value of MSGP is non-NIL, the MESSAGE from the ERROR call is displayed upon both the standard output device and the currently selected output device unless the standard output device is not open. The message appears prefixed with 5 asterisks. The MESSAGE list is displayed without top level parentheses. The MESSAGE from the ERROR call will be available in the global variable EMSG!*. The exact format of error messages generated by Standard LISP functions described in this document are not fixed and should not be relied upon to be in any particular form. Likewise, error numbers generated by Standard LISP functions are implementation dependent.

If no error occurs during the evaluation of U, the value of (LIST (EVAL U)) is returned.

If an error has been signaled and the value of TR is non-NIL a traceback sequence will be initiated on the selected output device. The traceback will display information such as unbindings of FLUID variables, argument lists and so on in an implementation dependent format.

3.9 Vectors

Vectors are structured entities in which random elements may be accessed with an integer index. A vector has a single dimension. Its maximum size is determined by the implementation and available space. A suggested input “vector notation” is defined in “Classes of Primitive Data Types”, section 2.2 on page 5 and output with EXPLODE, “Identifiers” section 3.3 on page 12.

GETV(**V**:*vector*, **INDEX**:*integer*):*any* *eval, spread*

Returns the value stored at position INDEX of the vector V. The type mismatch error may occur. An error occurs if the INDEX does not lie within 0...UPBV(V) inclusive:

***** INDEX subscript is out of range

MKVECT(**UPLIM**:*integer*):*vector* *eval, spread*

Defines and allocates space for a vector with UPLIM+1 elements accessed as 0..UPLIM. Each element is initialized to NIL. An error will occur if UPLIM is < 0 or there is not enough space for a vector of this size:

```
***** A vector of size UPLIM cannot be allocated
```

PUTV(**V**:*vector*, **INDEX**:*integer*, **VALUE**:*any*):*any* *eval, spread*

Stores VALUE into the vector V at position INDEX. VALUE is returned. The type mismatch error may occur. If INDEX does not lie in 0..UPBV(V) an error occurs:

```
***** INDEX subscript is out of range
```

UPBV(**U**:*any*):*NIL, integer* *eval, spread*

Returns the upper limit of U if U is a vector, or NIL if it is not.

3.10 Boolean Functions and Conditionals

AND([**U**:*any*]):*extra-boolean* *noeval, nospread*

AND evaluates each U until a value of NIL is found or the end of the list is encountered. If a non-NIL value is the last value it is returned, or NIL is returned.

```
FEXPR PROCEDURE AND(U);
BEGIN
  IF NULL U THEN RETURN NIL;
LOOP: IF NULL CDR U THEN RETURN EVAL CAR U
      ELSE IF NULL EVAL CAR U THEN RETURN NIL;
      U := CDR U;
      GO LOOP
END;
```

`COND([U:cond-form]):any` *noeval, nospread*

The antecedents of all U's are evaluated in order of their appearance until a non-NIL value is encountered. The consequent of the selected U is evaluated and becomes the value of the COND. The consequent may also contain the special functions GO and RETURN subject to the restraints given for these functions in "Program Feature Functions", section 3.7 on page 18. In these cases COND does not have a defined value, but rather an effect. If no antecedent is non-NIL the value of COND is NIL. An error is detected if a U is improperly formed:

```
***** Improper cond-form as argument of COND
```

`NOT(U:any):boolean` *eval, spread*

If U is NIL, return T else return NIL (same as function NULL).

```
EXPR PROCEDURE NOT(U);
  U EQ NIL;
```

`OR([U:any]):extra-boolean` *noeval, nospread*

U is any number of expressions which are evaluated in order of their appearance. When one is found to be non-NIL it is returned as the value of OR. If all are NIL, NIL is returned.

```
FEXPR PROCEDURE OR(U);
BEGIN SCALAR X;
LOOP: IF NULL U THEN RETURN NIL
      ELSE IF (X := EVAL CAR U) THEN RETURN X;
      U := CDR U;
      GO LOOP
END;
```

3.11 Arithmetic Functions

Conversions between numeric types are provided explicitly by the FIX and FLOAT functions and implicitly by any multi-parameter arithmetic function which receives mixed types of arguments. A conversion from fixed to floating point numbers may result in a loss of precision without a warning message being generated. Since integers may have a greater magnitude than that permitted for floating numbers, an error may be signaled when the attempted conversion cannot be done. Because the magnitude of integers is unlimited the conversion of a floating point number to a fixed number is always possible, the only loss of precision being the digits to the right of the decimal point which are truncated. If a function receives mixed types of arguments the general rule will have the fixed numbers converted to floating before arithmetic operations are performed.

In all cases an error occurs if the parameter to an arithmetic function is not a number:

***** XXX parameter to FUNCTION is not a number

XXX is the value of the parameter at fault and FUNCTION is the name of the function that detected the error. Exceptions to the rule are noted where they occur.

ABS(U:number):number *eval, spread*

Returns the absolute value of its argument.

```
EXPR PROCEDURE ABS(U);
  IF LESSP(U, 0) THEN MINUS(U) ELSE U;
```

ADD1(U:number):number *eval, spread*

Returns the value of U plus 1 of the same type as U (fixed or floating).

```
EXPR PROCEDURE ADD1(U);
  PLUS2(U, 1);
```

DIFFERENCE(U:number, V:number):number *eval, spread*

The value U - V is returned.

DIVIDE(U:number, V:number):dotted-pair *eval, spread*

The dotted-pair (quotient . remainder) is returned. The quotient part is computed the same as by QUOTIENT and the remainder the same as by REMAINDER. An error occurs if division by zero is attempted:

***** Attempt to divide by 0 in DIVIDE

```
EXPR PROCEDURE DIVIDE(U, V);
  (QUOTIENT(U, V) . REMAINDER(U, V));
```

EXPT(U:number, V:integer):number *eval, spread*

Returns U raised to the V power. A floating point U to an integer power V does not have V changed to a floating number before exponentiation.

FIX(*U: number*):*integer* *eval, spread*

Returns an integer which corresponds to the truncated value of U.
The result of conversion must retain all significant portions of U. If U is an integer it is returned unchanged.

FLOAT(*U: number*):*floating* *eval, spread*

The floating point number corresponding to the value of the argument U is returned. Some of the least significant digits of an integer may be lost do to the implementation of floating point numbers. **FLOAT** of a floating point number returns the number unchanged. If U is too large to represent in floating point an error occurs:

```
***** Argument to FLOAT is too large
```

GREATERP(*U: number, V: number*):*boolean* *eval, spread*

Returns T if U is strictly greater than V, otherwise returns NIL.

LESSP(*U: number, V: number*):*boolean* *eval, spread*

Returns T if U is strictly less than V, otherwise returns NIL.

MAX([*U: number*]):*number* *noeval, nospread, or macro*

Returns the largest of the values in U. If two or more values are the same the first is returned.

```
MACRO PROCEDURE MAX(U);
  EXPAND(CDR U, 'MAX2);
```

MAX2(*U: number, V: number*):*number* *eval, spread*

Returns the larger of U and V. If U and V are the same value U is returned (U and V might be of different types).

```
EXPR PROCEDURE MAX2(U, V);
  IF LESSP(U, V) THEN V ELSE U;
```

MIN([*U: number*]):*number* *noeval, nospread, or macro*

Returns the smallest of the values in U. If two or more values are the same the first of these is returned.

```
MACRO PROCEDURE MIN(U);
  EXPAND(CDR U, 'MIN2);
```

`MIN2(U:number, V:number):number` *eval, spread*

Returns the smaller of its arguments. If U and V are the same value, U is returned (U and V might be of different types).

```
EXPR PROCEDURE MIN2(U, V);
  IF GREATERP(U, V) THEN V ELSE U;
```

`MINUS(U:number):number` *eval, spread*

Returns -U.

```
EXPR PROCEDURE MINUS(U);
  DIFFERENCE(0, U);
```

`PLUS([U:number]):number` *noeval, nospread, or macro*

Forms the sum of all its arguments.

```
MACRO PROCEDURE PLUS(U);
  EXPAND(CDR U, 'PLUS2);
```

`PLUS2(U:number, V:number):number` *eval, spread*

Returns the sum of U and V.

`QUOTIENT(U:number, V:number):number` *eval, spread*

The quotient of U divided by V is returned. Division of two positive or two negative integers is conventional. When both U and V are integers and exactly one of them is negative the value returned is the negative truncation of the absolute value of U divided by the absolute value of V. An error occurs if division by zero is attempted:

```
***** Attempt to divide by 0 in QUOTIENT
```

`REMAINDER(U:number, V:number):number` *eval, spread*

If both U and V are integers the result is the integer remainder of U divided by V. If either parameter is floating point, the result is the difference between U and V*(U/V) all in floating point. If either number is negative the remainder is negative. If both are positive or both are negative the remainder is positive. An error occurs if V is zero:

```
***** Attempt to divide by 0 in REMAINDER
```

```
EXPR PROCEDURE REMAINDER(U, V);
  DIFFERENCE(U, TIMES2(QUOTIENT(U, V), V));
```

SUB1(U:number):number *eval, spread*

Returns the value of U less 1. If U is a FLOAT type number, the value returned is U less 1.0.

```
EXPR PROCEDURE SUB1(U);
  DIFFERENCE(U, 1);
```

TIMES([U:number]):number *noeval, nospread, or macro*

Returns the product of all its arguments.

```
MACRO PROCEDURE TIMES(U);
  EXPAND(CDR U, 'TIMES2);
```

TIMES2(U:number, V:number):number *eval, spread*

Returns the product of U and V.

3.12 MAP Composite Functions

MAP(X:list, FN:function):any *eval, spread*

Applies FN to successive CDR segments of X. NIL is returned.

```
EXPR PROCEDURE MAP(X, FN);
  WHILE X DO << FN X; X := CDR X >>;
```

MAPC(X:list, FN:function):any *eval, spread*

FN is applied to successive CAR segments of list X. NIL is returned.

```
EXPR PROCEDURE MAPC(X, FN);
  WHILE X DO << FN CAR X; X := CDR X >>;
```

MAPCAN(X:list, FN:function):any *eval, spread*

A concatenated list of FN applied to successive CAR elements of X is returned.

```
EXPR PROCEDURE MAPCAN(X, FN);
  IFNULL X THEN NIL
  ELSE NCONC(FN CAR X, MAPCAN(CDR X, FN));
```

MAPCAR(X:list, FN:function):any *eval, spread*

Returned is a constructed list of FN applied to each CAR of list X.

```
EXPR PROCEDURE MAPCAR(X, FN);
  IFNULL X THEN NIL
  ELSE FN CAR X . MAPCAR(CDR X, FN);
```

MAPCON(X:list, FN:function):*any* *eval, spread*

Returned is a concatenated list of FN applied to successive CDR segments of X.

```
EXPR PROCEDURE MAPCON(X, FN);
  IFNULL X THEN NIL
  ELSE NCONC(FN X, MAPCON(CDR X, FN));
```

MAPLIST(X:list, FN:function):*any* *eval, spread*

Returns a constructed list of FN applied to successive CDR segments of X.

```
EXPR PROCEDURE MAPLIST(X, FN);
  IFNULL X THEN NIL
  ELSE FN X . MAPLIST(CDR X, FN);
```

3.13 Composite Functions

APPEND(U:list, V:list):*list* *eval, spread*

Returns a constructed list in which the last element of U is followed by the first element of V. The list U is copied, V is not.

```
EXPR PROCEDURE APPEND(U, V);
  IFNULL U THEN V
  ELSE CAR U . APPEND(CDR U, V);
```

ASSOC(U:*any*, V:*alist*):{*dotted-pair*, NIL} *eval, spread*

If U occurs as the CAR portion of an element of the alist V, the dotted-pair in which U occurred is returned, else NIL is returned. ASSOC might not detect a poorly formed alist so an invalid construction may be detected by CAR or CDR.

```
EXPR PROCEDURE ASSOC(U, V);
  IF NULL V THEN NIL
  ELSE IF ATOM CAR V THEN
    ERROR(000, LIST(V, "is a poorly formed alist"))
  ELSE IF U = CAAR V THEN CAR V
  ELSE ASSOC(U, CDR V);
```

DEFLIST(**U**:*dlist*, **IND**:*id*):*list* *eval, spread*

A "dlist" is a list in which each element is a two element list: (ID:id PROP:any). Each ID in U has the indicator IND with property PROP placed on its property list by the PUT function. The value of DEFLIST is a list of the first elements of each two element list. Like PUT, DEFLIST may not be used to define functions.

```
EXPR PROCEDURE DEFLIST(U, IND);
  IF NULL U THEN NIL
  ELSE << PUT(CAAR U, IND, CADAR U);
        CAAR U >> . DEFLIST(CDR U, IND);
```

DELETE(**U**:*any*, **V**:*list*):*list* *eval, spread*

Returns V with the first top level occurrence of U removed from it.

```
EXPR PROCEDURE DELETE(U, V);
  IF NULL V THEN NIL
  ELSE IF CAR V = U THEN CDR V
  ELSE CAR V . DELETE(U, CDR V);
```

DIGIT(**U**:*any*):*boolean* *eval, spread*

Returns T if U is a digit, otherwise NIL.

```
EXPR PROCEDURE DIGIT(U);
  IF MEMQ(U, '(!0 !1 !2 !3 !4 !5 !6 !7 !8 !9))
  THEN T ELSE NIL;
```

LENGTH(**X**:*any*):*integer* *eval, spread*

The top level length of the list X is returned.

```
EXPR PROCEDURE LENGTH(X);
  IF ATOM X THEN 0
  ELSE PLUS(1, LENGTH CDR X);
```

LITER(**U**:*any*):*boolean* *eval, spread*

Returns T if U is a character of the alphabet, NIL otherwise.^a

```
EXPR PROCEDURE LITER(U);
  IF MEMQ(U, '(!A !B !C !D !E !F !G !H !I !J !K !L !M
              !N !O !P !Q !R !S !T !U !V !W !X !Y !Z
              !a !b !c !d !e !f !g !h !i !j !k !l !m
              !n !o !p !q !r !s !t !u !v !w !x !y !z))
  THEN T ELSE NIL;
```

^aThe published report omits escape characters. These are required for both upper and lower case as some systems default to lower.

MEMBER(*A: any, B: list*):*extra-boolean* *eval, spread*

Returns NIL if A is not a member of list B, returns the remainder of B whose first element is A.

```
EXPR PROCEDURE MEMBER(A, B);
  IF NULL B THEN NIL
  ELSE IF A = CAR B THEN B
  ELSE MEMBER(A, CDR B);
```

MEMQ(*A: any, B: list*):*extra-boolean* *eval, spread*

Same as MEMBER but an EQ check is used for comparison.

```
EXPR PROCEDURE MEMQ(A, B);
  IF NULL B THEN NIL
  ELSE IF A EQ CAR B THEN B
  ELSE MEMQ(A, CDR B);
```

NCONC(*U: list, V: list*):*list* *eval, spread*

Concatenates V to U without copying U. The last CDR of U is modified to point to V.

```
EXPR PROCEDURE NCONC(U, V);
BEGIN SCALAR W;
  IF NULL U THEN RETURN V;
  W := U;
  WHILE CDR W DO W := CDR W;
  RPLACD(W, V);
  RETURN U
END;
```

PAIR(*U: list, V: list*):*alist* *eval, spread*

U and V are lists which must have an identical number of elements. If not, an error occurs (the 000 used in the ERROR call is arbitrary and need not be adhered to). Returned is a list where each element is a dotted-pair, the CAR of the pair being from U, and the CDR the corresponding element from V.

```
EXPR PROCEDURE PAIR(U, V);
  IF AND(U, V) THEN (CAR U . CAR V) . PAIR(CDR U, CDR V)
  ELSE IF OR(U, V) THEN ERROR(000,
    "Different length lists in PAIR")
  ELSE NIL;
```

REVERSE(U:list):list *eval, spread*

Returns a copy of the top level of U in reverse order.

```
EXPR PROCEDURE REVERSE(U);
BEGIN SCALAR W;
  WHILE U DO << W := CAR U . W;
              U := CDR U >>;
  RETURN W
END;
```

SASSOC(U:any, V:alist, FN:function):any *eval, spread*

Searches the alist V for an occurrence of U. If U is not in the alist the evaluation of function FN is returned.

```
EXPR PROCEDURE SASSOC(U, V, FN);
  IF NULL V THEN FN()
  ELSE IF U = CAAR V THEN CAR V
  ELSE SASSOC(U, CDR V, FN);
```

SUBLIS(X:alist, Y:any):any *eval, spread*

The value returned is the result of substituting the CDR of each element of the alist X for every occurrence of the CAR part of that element in Y.

```
EXPR PROCEDURE SUBLIS(X, Y);
  IF NULL X THEN Y
  ELSE BEGIN SCALAR U;
          U := ASSOC(Y, X);
          RETURN IF U THEN CDR U
                  ELSE IF ATOM Y THEN Y
                  ELSE SUBLIS(X, CAR Y) .
                      SUBLIS(X, CDR Y)
        END;
```

SUBST(U:any, V:any, W:any):any *eval, spread*

The value returned is the result of substituting U for all occurrences of V in W.

```
EXPR PROCEDURE SUBST(U, V, W);
  IF NULL W THEN NIL
  ELSE IF V = W THEN U
  ELSE IF ATOM W THEN W
  ELSE SUBST(U, V, CAR W) . SUBST(U, V, CDR W);
```

3.14 The Interpreter

APPLY(FN:{*id,function*}, ARGS:*any-list*):*any* *eval, spread*

APPLY returns the value of FN with actual parameters ARGS. The actual parameters in ARGS are already in the form required for binding to the formal parameters of FN. Implementation specific portions described in English are enclosed in boxes.

```
EXPR PROCEDURE APPLY(FN, ARGS);
```

```
BEGIN SCALAR DEFN;
```

```
  IF CODEP FN THEN RETURN
```

```
    Spread the actual parameters in ARGS
    following the conventions: for calling
    functions, transfer to the entry point
    of the function, and return the value
    returned by the function.
```

```
  IF IDP FN THEN RETURN
```

```
    IF NULL(DEFN := GETD FN) THEN
```

```
      ERROR(000, LIST(FN, "is an undefined function"))
```

```
    ELSE IF CAR DEFN EQ 'EXPR THEN
```

```
      APPLY(CDR DEFN, ARGS)
```

```
    ELSE ERROR(000,
```

```
      LIST(FN, "cannot be evaluated by APPLY"));
```

```
  IF OR(ATOM FN, NOT(CAR FN EQ 'LAMBDA)) THEN
```

```
    ERROR(000,
```

```
    LIST(FN, "cannot be evaluated by APPLY"));
```

```
  RETURN
```

```
    Bind the actual parameters in ARGS to
    the formal parameters of the lambda
    expression. If the two lists are not
    of equal length then ERROR(000, "Number
    of parameters do not match"); The value
    returned is EVAL CADDR FN.
```

```
END;
```


EVAL(U:any):any *eval, spread*

The value of the expression U is computed. Error numbers are arbitrary. Portions of EVAL involving machine specific coding are expressed in English enclosed in boxes.

```

EXPR PROCEDURE EVAL(U);
BEGIN SCALAR FN;
  IF CONSTANTP U THEN RETURN U;
  IF IDP U THEN RETURN
    

|                                                                                                                         |
|-------------------------------------------------------------------------------------------------------------------------|
| U is an id. Return the value most currently bound to U or if there is no such binding: ERROR(000, LIST("Unbound:", U)); |
|-------------------------------------------------------------------------------------------------------------------------|


  IF PAIRP CAR U THEN RETURN
    IF CAAR U EQ 'LAMBDA THEN APPLY(CAR U, EVLIS CDR U)
    ELSE ERROR(000, LIST(CAR U,
      "improperly formed LAMBDA expression"))
    ELSE IF CODEP CAR U THEN
      RETURN APPLY(CAR U, EVLIS CDR U);
  FN := GETD CAR U;
  IF NULL FN THEN
    ERROR(000, LIST(CAR U, "is an undefined function"))
  ELSE IF CAR FN EQ 'EXPR THEN
    RETURN APPLY(CDR FN, EVLIS CDR U)
  ELSE IF CAR FN EQ 'FEXPR THEN
    RETURN APPLY(CDR FN, LIST CDR U)
  ELSE IF CAR FN EQ 'MACRO THEN
    RETURN EVAL APPLY(CDR FN, LIST U)
END;
```

EVLIS(U:any-list):any-list *eval, spread*

EVLIS returns a list of the evaluation of each element of U.

```

EXPR PROCEDURE EVLIS(U);
  IF NULL U THEN NIL
  ELSE EVAL CAR U . EVLIS CDR U;
```

EXPAND(L:list, FN:function):list *eval, spread*

FN is a defined function of two arguments to be used in the expansion of a MACRO. EXPAND returns a list in the form:

$$(FN L_0 (FN L_1 \dots (FN L_{n-1} L_n) \dots))$$

where n is the number of elements in L, L_i is the i th element of L.

```

EXPR PROCEDURE EXPAND(L, FN);
  IF NULL CDR L THEN CAR L
  ELSE LIST(FN, CAR L, EXPAND(CDR L, FN));
```

FUNCTION(FN:function):function *noeval, nospread*

The function FN is to be passed to another function. If FN is to have side effects its free variables must be fluid or global. FUNCTION is like QUOTE but its argument may be affected by compilation. We do not consider FUNARGs in this report.

QUOTE(U:any):any *noeval, nospread*

Stops evaluation and returns U unevaluated.

```
FEXPR PROCEDURE QUOTE(U);
  CAR U;
```

3.15 Input and Output

The user normally communicates with Standard LISP through “standard devices”. The default devices are selected in accordance with the conventions of the implementation site. Other input and output devices or files may be selected for reading and writing using the functions described herein.

CLOSE(FILEHANDLE:any):any *eval, spread*

Closes the file with the internal name FILEHANDLE writing any necessary end of file marks and such. The value of FILEHANDLE is that returned by the corresponding OPEN. The value returned is the value of FILEHANDLE. An error occurs if the file can not be closed.

```
***** FILEHANDLE could not be closed
```

EJECT():NIL *eval, spread*

Skip to the top of the next output page. Automatic EJECTs are executed by the print functions when the length set by the PAGE-LENGTH function is exceeded.

LINELENGTH(**LEN**:{*integer*, NIL}):*integer* *eval, spread*

If **LEN** is an integer the maximum line length to be printed before the print functions initiate an automatic TERPRI is set to the value **LEN**. No initial Standard LISP line length is assumed. The previous line length is returned except when **LEN** is NIL. This special case returns the current line length and does not cause it to be reset. An error occurs if the requested line length is too large for the currently selected output file or **LEN** is negative or zero.

***** LEN is an invalid line length

LPOSN():*integer* *eval, spread*

Returns the number of lines printed on the current page. At the top of a page, 0 is returned.

OPEN(**FILE**:*any*, **HOW**:*id*):*any* *eval, spread*

Open the file with the system dependent name **FILE** for output if **HOW** is EQ to **OUTPUT**, or input if **HOW** is EQ to **INPUT**. If the file is opened successfully, a value which is internally associated with the file is returned. This value must be saved for use by **RDS** and **WRS**. An error occurs if **HOW** is something other than **INPUT** or **OUTPUT** or the file can't be opened.

***** HOW is not option for OPEN

***** FILE could not be opened

PAGELLENGTH(**LEN**:{*integer*, NIL}):*integer* *eval, spread*

Sets the vertical length (in lines) of an output page. Automatic page EJECTs are executed by the print functions when this length is reached. The initial vertical length is implementation specific. The previous page length is returned. If **LEN** is 0, no automatic page ejects will occur.

POSN():*integer* *eval, spread*

Returns the number of characters in the output buffer. When the buffer is empty, 0 is returned.

PRINC(**U**:*id*):*id* *eval, spread*

U must be a single character id such as produced by EXPLODE or read by READCH or the value of !\$EOL!\$. The effect is the character U displayed upon the currently selected output device. The value of !\$EOL!\$ causes termination of the current line like a call to TERPRI.

PRINT(**U**:*any*):*any* *eval, spread*

Displays U in READ readable format and terminates the print line. The value of U is returned.

```
EXPR PROCEDURE PRINT(U);
  << PRIN1 U; TERPRI(); U >>;
```

PRIN1(**U**:*any*):*any* *eval, spread*

U is displayed in a READ readable form. The format of display is the result of EXPLODE expansion; special characters are prefixed with the escape character !, and strings are enclosed in "...". Lists are displayed in list-notation and vectors in vector-notation.

PRIN2(**U**:*any*):*any* *eval, spread*

U is displayed upon the currently selected print device but output is not READ readable. The value of U is returned. Items are displayed as described in the EXPLODE function with the exceptions that the escape character does not prefix special characters and strings are not enclosed in "...". Lists are displayed in list-notation and vectors in vector-notation. The value of U is returned.

RDS(**FILEHANDLE**:*any*):*any* *eval, spread*

Input from the currently selected input file is suspended and further input comes from the file named. FILEHANDLE is a system dependent internal name which is a value returned by OPEN. If FILEHANDLE is NIL the standard input device is selected. When end of file is reached on a non-standard input device, the standard input device is reselected. When end of file occurs on the standard input device the Standard LISP reader terminates. RDS returns the internal name of the previously selected input file.

```
***** FILEHANDLE could not be selected for input
```

READ():*any*

The next expression from the file currently selected for input. Valid input forms are: vector-notation, dot-notation, list-notation, numbers, function-pointers, strings, and identifiers with escape characters. Identifiers are interned on the OBLIST (see the INTERN function in "Identifiers", section 3.3 on page 12). READ returns the value of !\$EOF!\$ when the end of the currently selected input file is reached.

READCH():*id*

Returns the next interned character from the file currently selected for input. Two special cases occur. If all the characters in an input record have been read, the value of !\$EOL!\$ is returned. If the file selected for input has all been read the value of !\$EOF!\$ is returned. Comments delimited by % and end-of-line are not transparent to READCH.

TERPRI():**NIL**

The current print line is terminated.

WRS(FILEHANDLE:)*any**eval, spread*

Output to the currently active output file is suspended and further output is directed to the file named. FILEHANDLE is an internal name which is returned by OPEN. The file named must have been opened for output. If FILEHANDLE is NIL the standard output device is selected. WRS returns the internal name of the previously selected output file.

***** FILEHANDLE could not be selected for output

3.16 LISP Reader

An EVAL read loop has been chosen to drive a Standard LISP system to provide a continuity in functional syntax. Choices of messages and the amount of extra information displayed are decisions left to the implementor.

```

EXPR PROCEDURE STANDARD!-LISP();
BEGIN SCALAR VALUE;
  RDS NIL; WRS NIL;
  PRIN2 "Standard LISP"; TERPRI();
  WHILE T DO
    << PRIN2 "EVAL:"; TERPRI();
    VALUE := ERRORSET(QUOTE EVAL READ(), T, T);

```

```

        IF NOT ATOM VALUE THEN PRINT CAR VALUE;
        TERPRI() >>;
END;
```

QUIT()

Causes termination of the LISP reader and control to be transferred to the operating system.

4 System GLOBAL Variables

These variables provide global control of the LISP system, or implement values which are constant throughout execution.²

***COMP** = NIL *global*

The value of **!*COMP** controls whether or not PUTD compiles the function defined in its arguments before defining it. If **!*COMP** is NIL the function is defined as an xEXPR. If **!*COMP** is something else the function is first compiled. Compilation will produce certain changes in the semantics of functions particularly FLUID type access.

EMSG* = NIL *global*

Will contain the MESSAGE generated by the last ERROR call (see “Error Handling” section 3.8 on page 20).

\$EOF\$ = *<an uninterned identifier>* *global*

The value of **!\$EOF!** is returned by all input functions when the end of the currently selected input file is reached.

\$EOL\$ = *<an uninterned identifier>* *global*

The value of **!\$EOL!** is returned by READCH when it reaches the end of a logical input record. Likewise PRINC will terminate its current line (like a call to TERPRI) when **!\$EOL!** is its argument.

***GC** = NIL *global*

!*GC controls the printing of garbage collector messages. If NIL no indication of garbage collection may occur. If non-NIL various system dependent messages may be displayed.

²The published document does not specify that all these are GLOBAL.

NIL = NIL *global*
 NIL is a special global variable. It is protected from being modified by SET or SETQ.

***RAISE** = NIL *global*
 If !*RAISE is non-NIL all characters input through Standard LISP input/output functions will be raised to upper case. If !*RAISE is NIL characters will be input as is.

T = T *global*
 T is a special global variable. It is protected from being modified by SET or SETQ.

5 The Extended Syntax

Whenever it is possible to define Standard LISP functions in LISP the text of the function will appear in an extended syntax. These definitions are supplied as an aid to understanding the behavior of functions and not as a strict implementation guide. A formal scheme for the translation of extended syntax to Standard LISP is presented to eliminate misinterpretation of the definitions.

5.1 Definition

The goal of the transformation scheme is to produce a PUTD invocation which has the function translated from the extended syntax as its actual parameter. A rule has a name in brackets $\langle \dots \rangle$ by which it is known and is defined by what follows the meta symbol ::= . Each rule of the set consists of one or more “alternatives” separated by the | meta symbol, being the different ways in which the rule will be matched by source text. Each alternative is composed of a “recognizer” and a “generator” separated by the \implies meta symbol. The recognizer is a concatenation of any of three different forms. 1) Terminals - Upper case lexemes and punctuation which is not part of the meta syntax represent items which must appear as is in the source text for the rule to succeed. 2) Rules - Lower case lexemes enclosed in $\langle \dots \rangle$ are names of other rules. The source text is matched if the named rule succeeds. 3) Primitives - Lower case singletons not in brackets are names of primitives or primitive classes of Standard LISP. The syntax and semantics of the primitives are given in Part I.

The recognizer portion of the following rule matches an extended syntax procedure:

```
 $\langle function \rangle ::= ftype \text{PROCEDURE id } (\langle id \text{ list} \rangle);$   

 $\langle statement \rangle; \implies$ 
```

A function is recognized as an “ftype” (one of the tokens EXPR, FEXPR, etc.)

followed by the keyword **PROCEDURE**, followed by an “id” (the name of the function), followed by an *<id list>* (the formal parameter names) enclosed in parentheses. A semicolon terminates the title line. The body of the function is a *<statement>* followed by a semicolon. For example:

```
EXPR PROCEDURE NULL(X); EQ(X, NIL);
```

satisfies the recognizer, causes the generator to be activated and the rule to be matched successfully.

The generator is a template into which generated items are substituted. The three syntactic entities have corresponding meanings when they appear in the generator portion. 1) Terminals - These lexemes are copied as is to the generated text. 2) Rules - If a rule has succeeded in the recognizer section then the value of the rule is the result of the generator portion of that rule. 3) Primitives - When primitives are matched the primitive lexeme replaces its occurrence in the generator.

If more than one occurrence of an item would cause ambiguity in the generator portion this entity appears with a bracketed subscript. Thus:

```
<conditional> ::=
    IF <expression> THEN <statement1>
    ELSE <statement2> ...
```

has occurrences of two different *<statement>*s. The generator portion uses the subscripted entities to reference the proper generated value.

The *<function>* rule appears in its entirety as:

```
<function> ::= ftype PROCEDURE id (<id list>);<statement>; ==>
    (PUTD (QUOTE id)
      (QUOTE ftype)
      (QUOTE (LAMBDA (<id list>) <statement>)))
```

If the recognizer succeeds (as it would in the case of the NULL procedure example) the generator returns:

```
(PUTD (QUOTE NULL) (QUOTE EXPR) (QUOTE (LAMBDA (X) (EQ X NIL))))
```

The identifier in the template is replaced by the procedure name NULL, *<id list>* by the single formal parameter X, the *<statement>* by (EQ X NIL) which is the result of the *<statement>* generator. EXPR replaces ftype, the type of the defined procedure.

5.2 The Extended Syntax Rules

```
<function> ::= ftype PROCEDURE id (<id list>); <statement>; ==>
    (PUTD (QUOTE id)
      (QUOTE ftype)
      (QUOTE (LAMBDA (<id list>) <statement>)))
```


$\langle id\ list \rangle ::= id \implies id \mid$
 $id, \langle id\ list \rangle \implies id \langle id\ list \rangle \mid$
 $\implies NIL$

$\langle statement \rangle ::= \langle expression \rangle \implies \langle expression \rangle \mid$
 $\langle proper\ statement \rangle \implies \langle proper\ statement \rangle$

$\langle proper\ statement \rangle ::=$
 $\langle assignment\ statement \rangle \implies \langle assignment\ statement \rangle \mid$
 $\langle conditional\ statement \rangle \implies \langle conditional\ statement \rangle \mid$
 $\langle while\ statement \rangle \implies \langle while\ statement \rangle \mid$
 $\langle compound\ statement \rangle \implies \langle compound\ statement \rangle$

$\langle assignment\ statement \rangle ::= id := \langle expression \rangle \implies$
 $(SETQ\ id\ \langle expression \rangle)$

$\langle conditional\ statement \rangle ::=$
 $\mathbf{IF}\ \langle expression \rangle\ \mathbf{THEN}\ \langle statement_1 \rangle\ \mathbf{ELSE}\ \langle statement_2 \rangle \implies$
 $(COND (\langle expression \rangle\ \langle statement_1 \rangle) (T\ \langle statement_2 \rangle)) \mid$
 $\mathbf{IF}\ \langle expression \rangle\ \mathbf{THEN}\ \langle statement \rangle \implies$
 $(COND (\langle expression \rangle\ \langle statement \rangle))$

$\langle while\ statement \rangle ::= \mathbf{WHILE}\ \langle expression \rangle\ \mathbf{DO}\ \langle statement \rangle \implies$
 $(PROG\ NIL$
 $LBL\ (COND ((NULL\ \langle expression \rangle) (RETURN\ NIL)))$
 $\langle statement \rangle$
 $(GO\ LBL))$

$\langle compound\ statement \rangle ::=$
 $\mathbf{BEGIN\ SCALAR}\ \langle id\ list \rangle; \langle program\ list \rangle\ \mathbf{END} \implies$
 $(PROG (\langle id\ list \rangle) \langle program\ list \rangle) \mid$
 $\mathbf{BEGIN}\ \langle program\ list \rangle\ \mathbf{END} \implies$
 $(PROG\ NIL\ \langle program\ list \rangle) \mid$
 $\langle\langle\ \langle statement\ list \rangle\ \rangle\rangle \implies (PROGN\ \langle statement\ list \rangle)$

$\langle program\ list \rangle ::= \langle full\ statement \rangle \implies \langle full\ statement \rangle \mid$
 $\langle full\ statement \rangle\ \langle program\ list \rangle \implies$
 $\langle full\ statement \rangle\ \langle program\ list \rangle$

$\langle full\ statement \rangle ::= \langle statement \rangle \implies \langle statement \rangle \mid id: \implies id$

$\langle statement\ list \rangle ::= \langle statement \rangle \implies \langle statement \rangle \mid$
 $\langle statement \rangle; \langle statement\ list \rangle \implies$
 $\langle statement \rangle\ \langle statement\ list \rangle$

$\langle expression \rangle ::=$
 $\langle expression_1 \rangle . \langle expression_2 \rangle \implies$
 $(CONS\ \langle expression_1 \rangle\ \langle expression_2 \rangle) \mid$

$\langle expression_1 \rangle = \langle expression_2 \rangle \implies$
 (EQUAL $\langle expression_1 \rangle$ $\langle expression_2 \rangle$) |
 $\langle expression_1 \rangle \mathbf{EQ} \langle expression_2 \rangle \implies$
 (EQ $\langle expression_1 \rangle$ $\langle expression_2 \rangle$) |
 $\prime \langle expression \rangle \implies$ (QUOTE $\langle expression \rangle$) |
 function $\langle expression \rangle \implies$ (function $\langle expression \rangle$) |
 function($\langle argument list \rangle$) \implies (function $\langle argument list \rangle$) |
 number \implies number |
 id \implies id

$\langle argument list \rangle ::=$ () \implies |
 $\langle expression \rangle \implies \langle expression \rangle$ |
 $\langle expression \rangle, \langle argument list \rangle \implies \langle expression \rangle \langle argument list \rangle$

Notice the three infix operators . EQ and = which are translated into calls on CONS, EQ, and EQUAL respectively. Note also that a call on a function which has no formal parameters must have () as an argument list. The QUOTE function is abbreviated by '.

References

- [1] Computation Center. *LISP Reference Manual, CDC-6000*. The University of Texas at Austin.
- [2] Stanford Center for Information Processing. *LISP/360 Reference Manual*. Stanford University.
- [3] M. L. Griss and A. C. Hearn. A portable LISP compiler. *Software—Practice and Experience*, 11:541–605, June 1981.
- [4] A. C. Hearn. Standard LISP. *SIGPLAN Notices*, 4:28–49, 1969. Reprinted in SIGSAM Bulletin, ACM, Vol. 13, 1969, p. 28-49.
- [5] A. C. Hearn. REDUCE user’s manual: Version 3.3. Publication CP78 (Rev 1/88), RAND, 1988.
- [6] *MACLISP Reference Manual*, March 1976.
- [7] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmers Manual*. The M.I.T. Press, Cambridge, Massachusetts, 1965.
- [8] Mats Nordstrom, Erik Sandewall, and Diz Breslow. *LISP F1: A FORTRAN Implementation of LISP 1.5*. Uppsala University, Department of Computer Sciences.
- [9] Lynn H. Quam and Whitfield Diffie. *Stanford LISP 1.6 Manual*. Stanford Artificial Intelligence Laboratory, operating note 28.7 edition.
- [10] Warren Teitelman. *INTERLISP Reference Manual*. XEROX, Palo Alto Research Centers, 3333 Coyote Road, Palo Alto, California 94304, 1978.