LISP/370 SYNTAX AND SEMANTICS

LISP is almost unique among the programming languages, in that only a rather small kernel of knowledge is required to "understand" the meaning of its utterances. The whole question of "understanding" can be stated as: How does an expression of the following form evaluate? In all natural languages and most computer languages understanding through simple deduction is simply out of the question.

The understanding of LISP evaluation is possible through the mastery of about 30 rules, and a dictionary of primitive operators. Such an understanding is only sufficient in a limited but meaningful sense. Questions about the intent of a program or certain global understandings may not be revealed by this process.

In LISP we can deduce both the value denoted by an expression and also the sequence of computational states that its evaluation entails.

The three primitive expression classes are constants, variables, and combinations of expressions.

A LISP expression *e* is one of:

c a constant,
id a variable name or identifier.
(*rator rand*...) a *combination*
where the operator *rator* is an *e* and each operand *rand* is an *e*.

CONSTANTS

The evaluation of constants is trivial. Constants are idempotent, i.e. they evaluate to themselves.

The reserved identifier NIL is considered a constant.

Binary Programs

Binary programs and state descriptors are LISP/370 data objects and are considered constants. The application of these constants has special semantics.

A *bpi* is a binary program image object which is:
An *mbpi*, a machine language macro, or an *fbpi*, a machine language function.

The semantics of applying a binary program image *bpi* that was compiled from a defining expression is similar to the interpreted semantics of applying the expression. Any dissimilarity is due to compile time evaluation of macros, and operators that have "understood" meanings. The ultimate meaning results from the execution of the machine instructions. The semantics of *bpis* that are defined at the assembly language level is beyond the scope of this description.

State Descriptors

A state descriptor *sd* is a special type of constant. It is created by the understood operator STATE and certain understood meta operators that form *funargs*. It denotes or "captures" the state in which the STATE operator was applied. The computational state captured is, in essence, sufficient data to allow the multiple continuation of the computation. The current state of all memory settings is not included. Because of side effects (attributable to the imperative features of LISP) multiple continuations of a state may not all behave alike.

The exact nature of this data structure is not exposed. It does necessarily contain a component sufficient to define the environment ***E*** (see below).

A *funarg* is a form that combines an expression *e* with an environment thus giving meaning to the free variables of *e*. Such a form is a closed expression (closure).

An *sd* may be applied causing the saved state to continue. In that case the value of the STATE operator is some data value (and, by convention, usually not a saved state). In other words, the operator STATE gives an *sd* as value when saving, and some other message value if continuing.

VARIABLES

The value of a variable is defined by the current context or environment ***E***. We may view ***E*** as a function that maps an identifier into the place or ***binding*** in which the denoted value resides. ***E*** is a metalinguistic construct of this description of LISP/370 and not a first class data object. Non-the-less there are first class data objects that have (by implication) an ***E*** as a component. The metalinguistic access and update operators on the one component, stored objects, called ***bindings***, are built into the language. They are represented in the evaluation of a variable and the assignment operator.

Every evaluation takes place with respect to some context, and some evaluations create new ones. In particular, the application of abstractions creates a new context by augmenting the current context with new bindings for some identifiers. These new bindings take precedence over any former bindings of the same variables.

A feature of LISP/370 is the two classes of bindings that may be created. A fluid binding is accessible to any variable evaluation for which it is the most recent binding of the variable. A lexical binding is not accessible to called programs.

COMBINATIONS

Except for constants and variables, every expression is a combination. The combination form is used to indicate operator application. Some of these combinations are distinguished for semantic reasons.

There are three types of application expressions:

1. Meta combinations, a transformation from an operator value which is an *mr* and the list of unevaluated operands (*rand...*), which produces a data value.

2. Macro composition, a transformation from an operator value which is a macro and the original combination's data structure, (*rator rand...*), which produces a new expression. A macro is either a *mbpi*, or a *mlambda-exp*, or a closure of either of these.

3. Ordinary applications, a transformation from operator value and a list of the values of the operands, which produces a data value. An ordinary application results in the loss of access to the lexical variables of the current context. Ordinary application is presumed if neither of the other cases apply. If the operator is not recognizably applicable or inapplicable it is reevaluated and that value is ordinary applied.

The type of application depends on the value of the operator (it could be considered unfortunate that each type of application is not distinctly represented). The lack of transparency that results from using value rather than syntax to classify these application expressions is balanced by the flexibility of the delayed interpretation that can also be considered a feature of this LISP. Indeed the lack of distinction makes the definition of most operators a free choice between macro definition and ordinary function definition.

Macro composition example:

(PLUS 1 2 3) → (PLUS* 1 (PLUS* 2 3)) = 6

Meta combination example:

(QUOTE (FOO BAR)) = (FOO BAR)

META COMBINATIONS

The following identifiers constitute the class of basic understood meta operators *mrs*, i.e. their application as meta combinations is understood:

COND	LAMBDA
EXIT	MLAMBDA
FR*CODE	QUOTE
FUNARG	RETURN
FUNCTION	SEQ
GO	SETQ
LABEL	

The required syntax and semantics for these built in meta operators is as follows:

FUNARG

(FUNARG *e sd*) is a *funarg* or expression closure.

Semantics: The value of the *funarg* is the value of *e* evaluated with respect to the environment of *sd*.

Closure application also takes place with respect to the environment of the closure *sd*.

The Abstractions LAMBDA, MLAMBDA and FR*CODE

An *abstraction-exp* may be

a *lambda-exp* (LAMBDA *bv body*) †

where *bv*, the bound-variable part is

{*c* | (FLUID *id*) | (LEX *id*) | *id* | (bv_1 . bv_2) }

and *body* is an *e*.

---

† The notation (LAMBDA ...) is used for the case:
(e_1 ...) where the value of e_1 is LAMBDA; similarly for the other basic operators.

Semantics:
Evaluates to (FUNARG (LAMBDA *bv body*) *sd*)
where *sd* captures the current context, including lexical bindings.

This closure is ordinary-applicable to a list of values. A *lambda-exp* also is ordinary-applicable as are *fbpi*. The use of the word funarg to describe these closures stems from their early usage as functional arguments.

The meaning of the application of a closed *lambda-exp* to argument values is obtained by evaluating *body* in the context of *sd* augmented with the bindings formed by the conformation of *bv* onto the list of values.

Conformation consists of pairing components of the value list with the corresponding variable declaration in *bv*.

For variables named in the bound-variable part, the variable-declaration form (FLUID *id*) is required if other than lexical access is to be permitted.

In the case of the application of a *lambda-expression* that is not closed the current state provides the initial context. This is equivalent to reevaluating the *lambda-expression* and applying the resulting *funarg*.

MLAMBDA

An *abstraction-exp* may be

an *mlambda-exp* (MLAMBDA *bv body*)

Semantics:
Evaluates to (FUNARG (MLAMBDA *bv body*) *sd*)
where *sd* captures the current context.

This object or closure is macro-applicable to an argument. A *mlambda-exp* is also macro-applicable as are *mbpi*. For the case of macro composition this argument is the *combination* form whose operator value is the macro.

The meaning the macro-application of a closed *mlambda-exp* is obtained by evaluating *body* in the environment context of *sd* augmented with the bindings formed by the conformation of *bv* onto the argument.

In the case of the macro-application of a *mlambda-expression* that is not closed the current state provides the initial context.

The macro composition of such a combination entails the subsequent evaluation of the resulting form.

FR*CODE

An *abstraction-exp* may be an *operator-code-exp*

(FR*CODE e_2 *f-list lap-code*)

Semantics: As if e_2 were written instead. The semantics of this expression when compiled

QUOTE

(QUOTE *s-exp*) is a quoted s-expression.

Semantics: Evaluates to *s-exp*, the object quoted. Symbolic-expression, also s-expression and s-exp, are all terms used to denote the class of LISP data objects.

"...quotations play a role analogous to Gödel numbers in other formal theories." (Morris)†

SETQ

(SETQ *id* e_2), is an explicit assignment.

Semantics: Updates the current binding of *id*, $\boldsymbol{E}\{id\}$, with the value of e_2.

---

† James H. Morris, "Lambda-calculus Models of Programming Languages," PhD thesis, MAC-TR-57, Project MAC, Massachusetts Institute of Technology, Cambridge, Massachusetts, (1968) p. 35.

COND

(COND (*p* [*q*])...) is a conditional expression,

each predicate *p* is an *e*, and

each consequent *q* is an *e*.

Semantics: The predicates *p* of the predicate consequent clauses are evaluated sequentially until a non-NIL value is obtained. Then the consequent expression of that clause, if present, is evaluated as if it were written instead of the conditional-expression (side effects may have occurred); otherwise, the value is the value of the non-NIL predicate. If no clause has a non-NIL predicate, the value is NIL.

SEQ

(SEQ *s*...) is a statement-sequence-expression where

each statement *s* is a:

statement-label *tag* which is an *id*, or

program-statement *p-s*, an *e* which is not an *id*.

Semantics: Each *p-s* is evaluated in sequence in the statement context of the statement-labels. A statement context gives meaning to statement-labels. A statement-sequence that occurs as a program-statement (i.e. within another statement-sequence) will append its own context to that of the surrounding context. A statement-sequence that occurs in "expression context" creates only its own "statement context". The value of the statement-sequence is the value of the last *p-s*. Exit-expressions and go-statements can alter the normal sequence of evaluation.

GO

(GO *tag*) is a go-statement.

Semantics: If the go-statement occurs as a program-statement and the *tag* occurs in the context of the current statement-sequence-expression, then the sequential execution of program-statements proceeds with the statement following the *tag*, rather than the statement following the go-statement.

In the case where a go-statement occurs not as a program-statement but as an expression, then an "out of statement context GO error" results.

In the case that the *tag* does not occur in the current statement context, the go-statement is evaluated as though it were written instead of the current statement-sequence-expression (excepting that side effects may have occured). That is, one can go to the surrounding statement contexts so long as the current statement-sequence-expression was itself a program-statement *p-s*, etc.

EXIT

(EXIT { *id* | *ps* }) is an exit-expression.

Semantics: The main purpose of EXIT is to leave the current statement-label context. If the exit-expression occurs as other than a program-statement the value of the expression is the value of the operand. In the case that the exit-expression does occur in statement context:

Case 1: If the operand is a identifier it is treated as a variable and not a statement-label. The value of that variable becomes the value of the statement-sequence-expression.

Case 2: If the operand is not an identifier it is treated as though it were the last *p-s* of the current statement-sequence-expression. Note: (EXIT (GO A)) might not actually leave the current statement-label context if A were defined within the current statement context.

RETURN

(RETURN e_2) is a return-expression.

Semantics: e_2 is evaluated and its value becomes the value of the current ordinary application, or (EVAL *e sd*) expression, or macro application, whichever has the more immediate scope. RETURN returns its value back to the point at which the binding context of the environment ***E*** could possibly have been different.

EXIT takes control out of the current tag context (usually) and RETURN takes control out of the current binding context (contour).

FUNCTION

(FUNCTION e_2) is a closure-expression.

Semantics: Evaluates to (FUNARG e_2 *sd*) where *sd* captures the current state.

LABEL

(LABEL *bv body*) is a label-expression.

Semantics: The main purpose of this operator is to provide a way of denoting structures with cyclical references in them. This is important if you wish to define some functions that are closed with respect to some environment, and mutually recursive with respect to each other.‡

The following example was used by Steele and Sussman† to illustrate the LABELS operator in SCHEME. Here their example is rendered in the syntax and semantics of LISP/370. A global procedure COUNT counts the atoms of a tree structure sans terminal NILs. COUNT uses two local, closed, mutually recursive functions, namely COUNTCAR and COUNTCDR.

```
(SETQ COUNT
 (CDR
  (LABEL (COUNTCAR • COUNTCDR)
   (CONS
    (LAMBDA (L)
     (COND
      ( (ATOM L) 1)
      ( (PLUS (COUNTCAR (CAR L)) (COUNTCDR (CDR L)) )) ) )
    (LAMBDA (L)
     (COND
      ( (ATOM L) (COND ( (NULL L) 0) (1)))
      ( (PLUS (COUNTCAR (CAR L)) (COUNTCDR (CDR L)) )) ) ) ) ) ) )
```

Having evaluated the *body* of the LABEL expression with respect to an environment in which the elements of *bv* were bound to dummy pairs, those pairs are updated under the assumption that the value of *body* is an object of the same shape as *bv*. In the example we rely on the list nature of the funargs produced by the lambda-expressions.

For those interested in mathematical logic, we can make the allusion that label-expressions are a programmer's approximation to **Y**, the general fixed point finding function. The programmer should not be overwhelmed by these allusions to mathematical logic. **Y** merely assures the mathematician of the existence of a solution to expressions defined by recursive equations.

---

† Guy Lewis Steele Jr. and Gerald Jay Sussman, "The Revised Report on SCHEME A Dialect of LISP", AI MEMO 452, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, (1978) p. 4.

BASIC FUNCTIONS

The following identifiers are recognized as basic understood function values *fr*, i.e. their application is understood:

APPLX	FLOATP	NUMBERP
ATOM	FRP	PAIRP
BITSTRINGP	GENSYMP	PLEXP
CALL	IDENTP	RPLACA
CAR	LINTP	RPLACD
CDR	LISTP	SET
CONS	MDEFX	SMINTP
EQ	MRP	STATE
EVA1	NTUPLEP	STATEP
EVAL	NULL	STRINGP
FIXP		VECP

Of the above, the following are ordinary applicable but defined by very special rules:

APPLX, CALL, EVA1, EVAL, MDEFX, SET, and STATE.

(APPLX *fn list*)

APPLX performs the ordinary application of its first operand value to the list of values that is the value of the second operand. Lexical variables are accessible during this application.

(CALL a_1 ... *fn*)

CALL applies the value of its last operand to the list of values formed by evaluating its earlier operands. Lexical variables are inaccessible during this application.

(MDEFX *fn form*)

MDEFX is like APPLX except it performs a macro-application. It does not reevaluate the resulting expression as is the case for evaluating combinations that are macro compositions.

(EVA1 *e*)

EVA1 evaluates its one operand value with respect to the current environment. Lexical variables are accessible during this evaluation. This weakens (to some extent) the degree of protection that lexical variables might otherwise enjoy. Future plans for plugging such lexical leaks are being considered.

(EVAL *e sd*)

EVAL evaluates its first operand value with respect to the context of the state which is the value of its second operand.

(SET a_1 a_2)

SET is like SETQ except it evaluates its first operand, which must have an identifier as value. Lexical variables are accessible for this assignment.

(STATE [*gloval* [*glolst*]])

STATE saves the current state, or a modified form of it in the case that optional arguments were supplied.

The modified form of the current state may differ only in the global environment ***gloE*** component of the environment ***E***. This component is exercised only when the normal components of ***E*** (the ***bindings*** created by the application of abstractions) have been exhausted during the search for the most recent binding of a variable. The ***gloE*** gives the default or global ***binding***.

The value is a state descriptor *sd* which denotes the state in which the STATE operator was applied.

The *sd* may be used as an argument to EVAL to provide the environment of that state as the binding context for the evaluation.

An *sd* may be applied causing the saved state to continue. In that case, the value of the STATE operator is some data value (and not the saved state). In other words, the operator STATE gives an *sd* as value when saving, and some other message value if continuing.

The optional arguments **gloval** and **glolst** describe the modifications to the ***gloE***.

A ***gloE*** is a special object with two components:
glonot, the not present prescription for this ***gloE***,
is a pair (*gloval* • *gloalo*) where
gloval is NIL or else a two argument function
from an *id* and the *glolst* of the current ***gloE***, to the *s-exp* value for that variable in this global environment.
gloalo is NIL or a three argument function
from *s-exp, id,* and *glolst* to *globnd* values. Often the side effect of updating *glolst* is accomplished.
glolst, the global data list structure environment, is
({*glodat* | *globnd*} • {*glolst* | *glotrm*}) ,
where *globnd*, the global binding, is a *pair* (*id* • *s-exp*),
where *glodat*, the global own data, is any *s-exp* which is not a *pair*,
where *glotrm*, the global environment terminator, is {NIL | *sd*}.

PRIMITIVE OPERATORS FOR DATA PROCESSING

THE SINE QUA NON

ATOM
CAR
CDR
CONS
EQ
RPLACA
RPLACD

These familiar operators are defined elsewhere. As the compiler and interpreter have special understandings about what it means to apply these, they are not completely redefinable. Other important primitives are defined by binary programs and are subject to redefinition by the user.

LISP/370 supports the following aggregate data types:

Reference vectors
Selector Structures
Character strings
Bit strings
Word vectors
Vectors of floating point numbers
Pairs (lists)

The fundamental operators for accessing, updating, allocating and type testing predicates are all provided. These operators are defined elsewhere.

READ and PRINT

(PRINT *e* [*stream*])

Causes the characters of the external or canonical representation of the value of its first argument to be written to a stream.

The value of print is the value of its first argument.

(READ [*stream*])

Causes the characters of one entire s-expression to be read from a stream.

The value of READ is an internal data structure which is access equivalent to the one represented by the characters.

ACCESS EQUIVALENCE

(EQUAL x y)

For composite arguments, EQUAL implements access-equivalent equality testing. This means that two structures are EQUAL if every part of one structure which can be reached by a composition of accessing functions is EQUAL to the corresponding part of the the other structure reached through the same composition of accessing functions. Intuitively, two structures are EQUAL if they denote the same (possibly infinite) tree.

UPDATE EQUIVALENCE

(UEQUAL x y)

This is a generalized update-equality testing function applicable to any LISP object in the same sense as EQUAL. It differs from EQUAL in that for two structures to be UEQUAL, not only must corresponding parts of the structures be EQUAL through the access functions, but there must be the same number of unique parts and, if any of these parts were to be updated in one structure, the result would be EQUAL to the result of performing the same update on the other structure.

Intuitively, two structures are UEQUAL if and only if they denote equivalent rooted directed graphs, i.e. if they denote EQUAL structures which also have the same acyclical and cyclical sharing structure.