RC 7771 (#33639) 7/18/79
Computer Science 34 pages

LISP/370 CONCEPTS AND FACILITIES

Fred W. Blair*

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

Abstract: This paper describes the basic kernel of the LISP/370 system. It presents: an informal definition of the syntax and semantics for LISP/370 expressions; brief descriptions of basic data_types and data operators; description of input-output; description of interrupts and traps.

INTRODUCTION

This document is intended as a brief description of the central concepts and facilities of LISP/370. It does not attempt to be a users guide and is not an implementation description.

The version of LISP/370 described here is one of a series of systems produced during the continuing development of LISP at the IBM Thomas J. Watson Research Center. This version was selected for submission as an Installed User Program because it has been used for more than a year at that site, during which time we feel that most major implementation errors have been detected and corrected. Our thoughts about several aspects of LISP have evolved since this system was devised, but implementation of these new ideas is still in an experimental stage.

LISP/370 SYNTAX AND SEMANTICS

LISP† is noteworthy among 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? Understanding utterances of natural languages and most computer languages through deduction is simply out of the question. Understanding through deduction is unproductive when the underlying rules or axioms are not known.

The understanding of LISP/370 evaluation is possible through the mastery of the following concepts and the aid of a dictionary of primitive operators. Questions about the intent of a program or certain global understandings may not be answered by this process.

In LISP/370 we can deduce both the value denoted by an expression and also the sequence of computational states that its evaluation entails.

† The particular dialect LISP/370 that is defined here corresponds to IBM licensed program 5796-PKL.

In this description we attempt to convey the underlying rules without using much formal notation. In describing syntax however we use a few conventions:

{ and } are used for metalinguistic grouping.
| is used to separate alternatives.
[and] are used to indicate optionality.
The ellipsis "..." is used to denote zero or more objects.

The syntax of LISP/370 is described in the well known list form. This is really just one of the concrete representations that one might select. If you relax your attention from the parenthesis to the italicized words you may see that an abstract syntax is also defined.

While the LISP/370 kernel is thought to exhibit an improved structure over many other LISP dialects, it undoubtably has some weaknesses. For instance, certain identifiers have reserved meanings when applied. The readers' criticism is invited.

EXPRESSIONS

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

A LISP expression *e* is one of:
c denoting a constant,
id an identifier denoting a variable,
(*rator rand...*) denoting an operator-operands *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.

LISP/370 has the following broad classes of constants:
{*decimal-number* | *applicative-constant* | *nil* | *ranked-array* | *selector-structure*}
The representation forms for these constants is described in the output canonical form section.

The reserved identifier NIL is considered a constant.

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

Binary Programs

A *bpi* is a binary program image object which is:

An *mbpi*, a machine language macro,
or an *fbpi*, a machine language function.

A *bpi* is either a product of the compiler or of the assembly language programmer. The second case is beyond the scope of this description. 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. The compiler works by transforming the origional expression into a new LISP expression, in the process performing macro expansion and dealing with certain operators in special ways. The interpreted semantics of this new expression may differ slightly from those of the origional expression. The new expression is then compiled into machine code which has identical semantics.

State Descriptors

A state descriptor *sd* is a special type of constant. It is created by the special operator STATE and certain meta operators that form *funargs*, and "captures" the state in which the STATE operator was applied. The computational state captured is, in essence, sufficient to allow the continuation of the computation, but does not include the current state of all memory settings. Because of the effects of updating shared memory structures, multiple continuations of a state may not all behave alike.

The exact nature of the data structure constituting an *sd* is not exposed. It necessarily contains 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*. We use the terms 'closure' and 'funarg' interchangably when refering to such forms. The use of the word funarg to describe these closures stems from their early usage as functional arguments.

Application of an *sd* causes the saved state to continue. In that case the *sd* is applied to an expression whose value appears as the value of the STATE operator (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. It is described is more detail later.

VARIABLES

The value of a variable is defined by the current context or environment ***E***. We may view ***E*** as a function that maps a variable into the place or ***binding*** in which its value resides. ***E*** is a metalinguistic construct of this description and not a LISP/370 data object. Nevertheless there are first class data objects that have (by implication) an ***E*** as a component.

Bindings are stored objects on which metalinguistic access and update operators are defined. Evaluation of a variable involves accessing the value in the appropiate binding, and assignment, SETQ, involves its replacement.

Every evaluation takes place with respect to some environment, and some evaluations create new ones. In particular, the application of an *abstraction* creates a new environment by augmenting the current one with new bindings for some identifiers; any former bindings of the same variables are superseded.

In LISP/370 two classes of bindings may be created. A fluid binding is accessible to any evaluation of a variable for which it is the most recent binding. A lexical binding is not accessible to CALLed or ordinary-applied operators and thus offers some degree of isolation from side effects.

Whenever no normal binding takes precedence, the global environment ***gloE*** is invoked to produce the global binding. The nature of ***gloE*** is rather ad hoc and flexible (see the STATE operator for more details). It is worth noting that the normal default ***gloE*** is such that variables have their denoting *id* as value until otherwise assigned.

A most significant aspect of LISP is the way that environments can be retained as data objects and dynamicly invoked. In LISP "referential transparency" is optional. Indeed, keeping track of the contexts can become a major preoccupation. LISP/370 is somewhat remiss in not offering the basic operators as constants with convenient notation; it relies on operator variables that evaluate to themselves.

COMBINATIONS

Except for constants and variables, every expression is a *combination*. *Combinations* are used to indicate operator application. Some combinations are distinguished for semantic reasons.

There are three types of *combinations*:

1. Meta-combination: A transformation from an operator value which is an *mr*, and the list of unevaluated operands (*rand...*), into a data value.
Meta combination example:
(QUOTE (FOO BAR)) = (FOO BAR)

2. Macro-composition: A transformation from an operator value which is a macro, and the original combination (*rator rand...*), into a new expression. A macro is an *mbpi*, or an *mlambda-exp*, or a closure of either of these.
Macro composition example:
(PLUS 1 2 3) → (PLUS* 1 (PLUS* 2 3)) = 6

3. Ordinary-application: A transformation from an operator value which is neither a macro nor an *mr*, and a list of the values of the operands, into a data value. An ordinary application results in the loss of access to the lexical bindings of the current context. If the operator is not recognizably applicable or inapplicable it is reevaluated and that value is ordinary-applied.

Thus, the type of application depends on the value of the operator. It could be considered unfortunate that each type of application is not represented by a distinct syntax. The resulting lack of transparency is balanced by the flexibility of the delayed interpretation that can 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.

META COMBINATIONS

The following identifiers constitute the class of basic meta operators *mr*, i.e. their application should be understood as meta-combination.

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

The syntax and semantics for these built in meta operators is now given.

FUNARG

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

Semantics:
The value of the *funarg* is the value of *e* evaluated in the environment of *sd*.

The application of the *funarg* is the application of its expression-part *e* in the environment of its *sd*. Free variables of *e* resolve to the bindings of the context in which the funarg was formed, including the lexical variables of that context.

† The notation (FUNARG *e sd*) is used for the case (e_1 *e sd*) where the value of e_1 is FUNARG; similarly for the other basic operators.

The Abstractions LAMBDA, MLAMBDA and FR*CODE:

An *abstraction-exp* may be a *lambda-exp* or an *mlambda-exp* or an *operator-code-exp*:

LAMBDA

(LAMBDA *bv body*) is a *lambda-exp*.

where *bv*, the bound-variable part is {*c* | (FLUID *id*) | (LEX *id*) | *id* | (bv_1 • bv_2) } and *body* is an *e*.

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*.

When a closed *lambda-exp* is applied to argument values, the meaning 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

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

Semantics:
Evaluates to (FUNARG (MLAMBDA *bv body*) *sd*), where *sd* captures the current context. This closure is macro-applicable to an argument. A *mlambda-exp* is also macro-applicable as are *mbpi*. In the case of macro-composition the argument is the *combination* form whose *rator* is the macro.

The meaning of the macro-application of a closed *mlambda-exp* is obtained by evaluating *body* in the environment 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-exp* that is not closed the current state provides the initial context.

A macro composition entails macro-application and the subsequent evaluation of the resulting form. The result of the macro-application is treated as though it were written instead of the macro composition form.

FR*CODE

(FR*CODE e_2 *f-list lap-code*) is an *operator-code-exp*.

Semantics:
When interpreted, as if e_2 were written instead; when compiled, as determined by the *lap-code*. The reader may view this as a window into the hell of assembly language semantics, which is offered to overcome the mistakes of the designers.

SETQ

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

Semantics:
Updates the current environment E so that the value of e_2 becomes the value component of $E\{id\}$, the binding of *id*.

FUNCTION

(FUNCTION e_2) is a closure-expression.

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

QUOTE

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

Semantics:
Evaluates to *s-exp*, a Symbolic-expression. This term (also s-expression), is used to denote the class of LISP data objects.

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

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 label-context

† 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.

to that of the surrounding context. A statement-sequence that occurs in "expression context" creates 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 occurred). 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)) would not 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 (unless its operand *ps* switchs control first) and RETURN takes control out of the current binding context (contour).

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.

---

† 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 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.

```
(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 with the corresponding components of the value of *body*. This is done 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. Thus the initial value of body was a pair of funarg lists; the dummy pair that was the value of COUNTCAR had its CAR component replaced with FUNARG and its CDR component replaced with ((LAMBDA ...) *sd*) from CDR of the first funarg. Similarly for COUNTCAR using the components of the second funarg list. The net result is a pair of funargs whose environments contain bindings with references to funargs that are access equivalent to themselves. Each function is closed with respect to an environment in which CARCOUNT and CDRCOUNT are both bound to the equivalent closed function.

BASIC FUNCTIONS

The following identifiers are recognized as basic function operators *fr*, i.e. their ordinary-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_l ... *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. An improved design should consider plugging such lexical leakage.

(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 [*glonot* [*glolst*]])

STATE captures the current state, or a modified form of it in the case that optional arguments were supplied. The value is a state descriptor *sd* which denotes the state in which the STATE operator was applied.

The modified form of the current state may differ only in the global environment ***gloE*** component of the environment ***E***. This component is used 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. Thus, ***gloE*** gives the ***global-binding***.

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

Application of an *sd* causes the saved state to continue. In that case the *sd* is applied to an expression whose value appears as the value of the STATE operator (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.

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

A ***gloE*** is a special object with two components:
glonot, the 'not present' prescription,
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 canonical representation of the value of its first argument to be written to a stream. The nature of streams will be described in a later section.

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

Canonical Representation

An *s-exp* is:

[*label*] {*c*ƀ | *id*ƀ | *combination* }
where *label* is {*label-name* = },
where *label-name* is {%L*digit*$_1$...*digit*$_n$} where $1 \le n \le 8$,
and *id* ∊ *ID* the set of identifiers (names),
and *c* ∊ *C* the set of constants,
and *combination* = (*comp*$^{+}_{\text{ƀ}}$ [ƀ • ƀ *comp*]) ‡
where *comp* is {*label-name* | *c* | *id* | *combination* | {*label comp*} }

Notice that the term *list* was avoided and *combination* was used instead. This may be helpful in avoiding the common misconception that LISP is just a list-processor!

One of the goals of LISP/370 is to enrich the class of data types supported. The output canonical syntaxes for constants and identifiers (names) are defined below.

A constant *c* is:

{*decimal-number* | *applicative-constant* | *nil* | *ranked-array* | *selector-structure*}
where *decimal-number* = {*integer* | *floating-point*}
where *integer* = [*sign*] $digit^+$
where *sign* = {+ | −}
where *digit* = {0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9}
where *floating-point* = *integer* • digit... [E *decimal-number*]
where *applicative-constant* = { *bpi* | *mr* | *fr* | *sd* }
where *bpi* = †
where *mr* = { LAMBDA | MLAMBDA | QUOTE | SETQ | FUNCTION | LABEL | COND | SEQ | GO | EXIT | PROGN | RETURN | FR*CODE }
where *fr* = {*fix-ur* | *mult-ur*}
where *fix-ur* = {EVA1 | MDEFX | APPLX | EVAL | SET | CLOSURE | ...}
Comment: Many more basic operators that take definite numbers of arguments will fall into this class.
where *mult-ur* = {STATE | CALL | ... }
Comment: Many more basic operators that take indefinite numbers of arguments will fall into this class.
where *sd* = (no syntactic form available or intended)
where *nil* = ()

‡ x^{*}_{b} is used to indicate zero or more *x* separated by blanks.

x^{+}_{b} is used to indicate one or more *x* separated by blanks.

~ is used as a metalinguistic set difference operator. *M*~*N* for the complement of *N* in *M*; all points of *M* not in *N*.

† Indicates not fully supported.

where *ranked-array* = {*vector* | *string* }

where *vector* = {*pointer-vector* | *intermediate-integer-vector* | *floating-point-vector*}

where *pointer-vector* = < *comp*$^{*}_{b}$ >

where *intermediate-integer-vector* = %I< *integer*$^{*}_{b}$ >

where *floating-point-vector* = %F < *floating-point*$^{*}_{b}$ >

where *string* = { *character-string* | *bit-string* }

where *character-string* =

{ ' *char*^{n} ' | %*chr-capacity* ' *char*^{n} ' }

where *chr-capacity*−n > 3

and *chr-capacity* = 1+4i where i ∈ {1 2 ...}

where *char* = {*chr* | |*anychr*}

where *chr* ∈ *CHR* = {*ANYCHR* ~ { ' |}}

where *ANYCHR* is the set of all characters available

where *bitstring* =

%B [*capacity*] { ' *hex** ' | :[*content-len*] ' *hex** ' }

where *capacity* = 8+32i where i ∈ {1 2 ...}

Comment: *capacity* is present if there is an excess of 32 bits over the *content-len*.

where *content-len* = *digit*$^{+}$

Comment: *content-len* is present if the number of bits contained is not a multiple of four.

where *hex* = {0|1|2|3|4|5|6|7|8|9|A|B|C|D|C|E|F}

where *selector-structure* = %S(*comp*$^{*}_{b}$ [ƀ • ƀ *bit-string*]) †

An *id* is: {*norid* | *gensym* }

where *norid* ∈ {x| {*non-num id-chr*...}}

where *non-num* ∈ { {{*ID-CHR* ~ *DIGIT*} ~ { % } } | |*anychr*}

where *id-chr* = {x∈{{*ANYCHR* ~ *IDDELIM*} ~ {|}} | |*anychr*}

where *iddelim* = {ƀ|(|)|<|>}

where *gensym* = %G*gennum*

where *gennum* = *digit*$^{+}$

(READ [*stream*])

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

---

† Indicates not fully supported.

The value of READ is an internal data structure which is update equivalent to the one represented by the characters, except for *gensyms*. Gensyms are never interned like the normal identifiers and READ preserves their EQ-ness only locally. For each gensym, unique to the s-expression representation that READ scans, a new internal gensym is allocated.

READ accepts representations in output canonical form but is prepared to be somewhat liberal with regard to blanks, content-len, chr-capacity, and dots.

ACCESS EQUIVALENCE

(EQUAL x y)

Two non-composite objects are EQUAL if they have the same canonical representation. 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, and that part can be reached through the same composition of accessing functions. Intuitively, two structures are EQUAL if they denote the same (possibly infinite) tree.

Caution: Two expressions that are EQUAL may not be computationally equivalent. For example:

(EQ (QUOTE %L1=(A))(QUOTE %L1)) is true in any context, and
(EQ (QUOTE (A))(QUOTE (A))) is false in any context.

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.

THE LISP/370 DESTRUCTIVE STREAM FACILITY

In Stoy and Strachey[1] *streams* were proposed as vehicles for the transfer of information in systems. In this system we copy their concept to a large extent.

Burge[2] has posited an even more abstract and general stream model in which *destructive streams* are a special case. We can recognize the Stoy and Strachey model as this interesting special case. A *destructive stream* is a stream which has private storage within itself which undergoes updating. This allows successive items of a stream to occupy the same storage. In Burge's more general model ordinary streams are applicable functions, are not 'destructive', are retainable and capable of backup. In his model "A *stream* is a functional analog of a coroutine [3, 4] and may be considered to be a particular method of representing a list in which the creation of each list element is delayed until it is actually needed.".

In LISP these abstract streams are definable, but we choose to suggest a data-structure model for streams and basic stream-data-structure manipulation facilities. We choose to use a data-structure, instead of using functionals, for reasons of efficiency and to allow updating. The stream data structure is simply a pair, the first element of which is the current item at the head of the sequence, the second element of which serves to define the rest of the sequence.

A *stream* is (*heads* • *rests*) where,
    heads is the next item of the stream which can be any *s-exp*, and
    rests serves to define the rest of the stream and is either:
        a *stream*,
        or a special stream description <*rfn bd asc* [*any*...]>,
            where *rfn* the stream dependent function is an *e*,
            and *bd* the fast-buffer description is NIL in the case of *slow-streams*,
                or < *string begindex curindex endindex*> for *fast-streams* of characters,
                    where *begindex* the beginning index is a {0 | 1 | ...},
                    and *curindex* the current character index is a {0 | 1 | ...},
                    and *endindex* the boundary index is a {0 | 1 | ...},
            and *asc* the *associated-states* which is an *a-list*,
            and *any* is any stream dependent information that the user provides,
        or *other*, a stream terminator which is any non-vector *atom*.

---

1. J. E. Stoy and C. Strachey. "OS6—An Experimental Operating System for a Small Computer." *Computer J.* **15**, No. 2. 117 and No. 3. 195 (1972)
2. W. H. Burge. "Stream Processing Functions" *IBM J. Res. Develop.* **19**, 12 (1975).
3. M. E. Conway. "Design of a Separable Transition-diagram Compiler." *Commun. ACM* **6**, 396 (1963).
4. A. Evans. "PAL—A Language Designed of Teaching Programming Linguistics." *Proc. 23rd ACM Conf.*, 395 (1968).

The basic primitives for streams are:

(EQ (CDR stream) stream)

This tests if 'stream' is the empty stream.

%L1=(%L1 • %L1) is a representation of an empty stream.

Thus, the empty stream is one which is incapable of emitting anything but the stream itself.

> Note: In this paper the labels used to convey EQ'ness have scope extending over the entire equation in which they are used.
> For example in: g{%L1=(a • b)} → %L1=(c • d) it is meant that %L1 is updated.

Consider the following fast stream which is not empty but is nearly so:
%L1=(%L1 • <rfn <string 0 n n> ()>) where a subsequent application of NEXT will produce %L1=(%L1 • %L1). The interpretation of this stream is that it is a unit stream with end-of-line as its last item. The stream itself will serve not only as stream terminator (as *rests*) but also as end-of-line indicator when it appears as *heads*.

(NEXT stream)

NEXT is a function from streams to streams, which for specific types of streams produces as value the argument stream updated. NEXT is most efficient for fast streams. The action of NEXT is defined by the following rules:

next{ %L1=(*heads* • {*other* | %L1}) } → %L1=(%L1 • %L1) .

next{ %L1=(x y • z)} → %L1=(y • z) .

next{%L1=(*heads* • <*rfn* () x ...>)} → *rfn*{%L1}.

next{%L1=(*heads* • %L2=<*rfn* %L3 x...>)}
where %L3=<*string begindex curindex endindex*>

→ *rfn*{%L1} if *curindex* $\geq$ *endindex*, and *heads* = %L1,

→ %L1=(%L1 . %L2) if *curindex* $\geq$ *endindex*, and *heads* $\neq$ %L1,
(This illustrates the production of end-of-line symbols)

otherwise → %L1=(y • %L2=<*rfn* %L3 x...>)
where L3=<*string begindex curindex*+1 *endindex*>
and y = fetchchar{*string* ; *curindex*} .

(WRITE s-exp stream)

write{x;%L1=(y • %L1)} → %L1=(x • NIL) .

write{x;%L1=(y • z)} → %L1=(x y • z)
where z is *other* or a *stream* that is not EQ to %L1.

write{x;%L1=(y • <*rfn* () z...>)} → *rfn*{x;%L1}.

write{x;%L1=(y • %L2=<*rfn* %L3 z...>)}
where %L3=<*string begindex curindex endindex*>

→ %L1=(x • %L2=<*rfn* %L3 z...>)
where %L3=<storechr{ *string*; *curindex*; x} *begindex curindex*+1 *endindex*>
if *curindex* $<$ *endindex*, and x is a *character*,

otherwise → *rfn*{x;%L1} .

(TEREAD stream)

teread{%L1=(%L1 • y)} → %L1=(%L1 • y) .

For x ≠ %L1 :

teread{%L1=(x • {*other*| %L1})} → %L1=(%L1 • %L1) .

teread{%L1=(x • *stream*)} → %L1=(%L1 • cdr{teread{*stream*}}) .

teread{%L1=(x • %L2=<*rfn* () x...>)} → %L1=(%L1 • %L2) .

teread{%L1=(x • *faststream*)} → %L1=(%L1 • *faststream*')
where *faststream*' is *faststream* with *curindex* updated to the value of *endindex*.

(TERPRI stream)

terpri{x} = terprix{x;x}.

terprix{x;%L1=(y • {*other* | %L1})} → x .

terprix{x;(y • *stream*)} → terprix{x; *stream*} .

terprix<x; %L1=(y • <*rfn* z...>)} → *rfn*{x;%L1} .

Some Distinguished Streams

LISPIT the console input stream. LISPIT is a fluid variable with the following initial value:

%L1=(%L1 • < lispitin < nil 0 0 0> asc NIL>)
where asc=((DEVICE • CONSOLE)(MODE • I)(QUAL • V)).

After the file is activated:

%L1=(item • < lispitin < string beg cur end>asc p-list>)
where p-list denotes a system dependent I/O control block, or NIL
and lispitin is an input console stream dependent function which is capable of activating the file when the p-list field contains NIL.

The function lispitin achieves system independency by special calls to system dependent portals for all system dependent computation. Activating this stream consists of:

1. Building an input console p-list in a system dependent manner.

2, Determining the console linelength (also system dependent) and allocating string, a lisp character vector used to provide an input area for the terminal line. The capacity of string is sufficient to hold the determined maximum input linelength, and its contents-length reveals how many it actually holds.

3. Initializing beg to 0, cur and end to linelength.

4. Applying lispitin to the now active stream.

When lispitin is applied to an active stream it causes a system dependent console input operation to refill string, resetting string-length to the actual number of characters read, setting end to that number also, and setting beg to zero and cur to one. If the number of characters read was zero the stream becomes:

%L1=(%L1 • <lispitin <'' 0 0 0> asc p-list>)
otherwise:
%L1=(c_0 • <lispitin <'$c_0 \ldots c_{end-1}$' 0 1 end> asc p-list>).

LISPOT the console output stream.

LISPOT is a fluid variable with the following initial value:
%L1=(%L1 • <lispotout <NIL 0 0 0> asc p-list>)
where asc=((DEVICE • CONSOLE)(MODE • O)), and p-list=NIL,
and lispotout is similar to lispitin except it needs less information to build the p-list.

After %L1 is activated by lispotout by write{c;%L1} it becomes:

%L1=(c • <lispotout < string 0 1 end> asc p-list>)
where end is the system dependent preferred console output line-length and string is 'c' . The capacity of string is end characters.

Where lispotout works in much the same manner as lispitin. One peculiarity of lispotout (and hopefully any output stream which is inactive) occurs when the initial write is in effect a TERPRI.

write{%L1;%L1=(%L1 • <lispotout <NIL 0 0 0> asc NIL>)}
→ %L1=(%L1 • <lispotout <string 0 0 end> asc NIL>)
where string= '' but has capacity for 'end' characters.

User Stream Definition Facilities

(DEFIOSTREAM asc linelen position)

DEFIOSTREAM produces as value a fast-stream which interfaces with the real input/output devices.

The actual stream produced is system dependent but the operation of saving a lisp system and bringing it up on another operating system entails the reactivation of all such streams; in which case they may become defined for the new system. The user would have to contrive to have the actual files moved and converted if that were necessary.

The parameters of DEFIOSTREAM are as follows:

'asc' is an a-list, *i.e.* (property...)
where property is:

{(FILE • {('fname' ['ftype' ['fmode']]) | 'dsname'}) |
(DEVICE • CONSOLE) } or,

(RECFM • {F | V}) or,

(MODE • {I | INPUT | O | OUTPUT}) or,

(QUAL •
if CONSOLE input then {S | T | U | V | X}
if CONSOLE output then {LIFO | FIFO | NOEDIT}

The value of the FILE property may be either character string, as indicated, or identifier, in which case the identifier pname is used.

'linelen' is linelength if required, else NIL. For input files, the user supplied linelen is passed to a system dependent portal and the portal gives back a number (possibly the same one) which is used as the actual size of the buffer string which is allocated at activation time. This parameter does not specify a truncation column. For output streams linelen determines both string size and end index.

'position' is a linenumber which defines the starting position if required else NIL.

What follows are some examples of operating system interface streams, their definition and use.

defiostream{asc;72;1}
    where asc = ((FILE XXX LISP)(RECFM • V)(MODE • I)).
→ %L1=(%L1 • <filein <NIL 0 72 72> asc NIL 1>)

Comment: Defines an input stream from the file system. The number 72 is the users idea of the length of the longest record. For most operating systems the actual file characteristics will take precedence.

next{%L1=(%L1 • <filein <NIL 0 72 72> asc NIL 1>)}
→ %L1=(c_0 • <filein <%120'$c_0 \ldots c_{99}$' 0 1 100> asc p-list 2>)

> where the string '$c_0 \ldots c_{99}$' in this instance has 100 characters but has a capacity for 120 characters because 120 was determined to be the actual longest record of the file.
>
> where p-list is a system dependent I/O control block designation and will not be explained.

This illustrates normal behavior of next when *curindex*$\geq$*endindex* and *heads* is the stream itself, and the line read in is not empty.

If the first line were empty:
next{%L1}→%L1=(%L1 • <filein <'' 0 0 0> asc p-list 2>)
and similarly for subsequent empty lines.

On end of file: %L1=(%L1 • %L1) .

defiostream{asc;72;1}
    where asc=((FILE YYY LIST)(RECFM • V)(MODE • O))

Comment: Defines an file system output stream. In the case that an old output file exists, its existence is to be ignored as much as possible. The longest record that we wish to write is 72 characters.

Initially the above definition gives rise to:

> %L1=(%L1 • <fileout <'' 0 0 72> asc NIL 1>) where '' has capacity for 72 characters.

```
write{c0;%L1=(%L1 • <fileout<'' 0 0 72>asc NIL 1>)}
→ %L1=(c0 • <fileout <'c0' 0 1 72> asc p-list 1>)
```

However,

```
write{%L1;%L1=(%L1 • <fileout<'' 0 0 72>asc NIL 1>)}
→ %L1=(%L1 • <fileout <'' 0 0 72> asc p-list 2>)
```

THE INTERRUPT SYSTEM

Interrupts are responses to external or asynchronous events. The event that causes the interrupt communicates this to LISP by updating some shared storage structures. LISP polls to see if any interrupt has occurred. It does this at times when it has a "clean state".

If an interrupt is pending a DISPATCHER is called. The interrupt service function is dispatched for the highest priority pending interrupt whose priority is greater than the current level of priority of the interrupted process.

The global variable EXTERNAL-EVENTS-CHANNELS has a value which is a vector whose k^{th} element is a function of no arguments, which should be the service function for interrupts of type k. See Table 1, for the detailed definition for each channel. This vector is a LISP reference vector and normal vector operations may be used on it, with caution!

The function S,ERRORLOOP is most commonly employed as the service function. Its LISP source is in ERROR LISP370 but a brief description is in order.

Basicly a READ, EVA1, PRINT loop. One can exit normally to resume the interrupted process by incanting (FIN *e*). One can do a UNWIND which is an non-local goto to the nearest error catcher; as S,ERRORLOOP itself has such a catcher one must signal it to do an UNWIND by (UNWIND *n*) where *n*>0. The action of UNWIND should reset the current priority level to 0 and turn the polling back on. Uncontrolled continuation (applying states) from high priority interrupts could cause loss of sensitivity to lower priority interrupts. S,ERRORLOOP1 is just like S,ERRORLOOP except it doesn't have it own error catcher. It is used when the system is seriously out of space.

Table 1. The External Events Channels

Channel Number	Service Expression Definition or Explanation
0	Not really a channel. Reserved for future use.
1	Currently, UNUSED-CHANNEL1 with priority 1.
2	EXTERNAL-INTERRUPT = (LAMBDA()(S,ERRORLOOP 16 "EXT" (STATE))) has priority 3.
3	ALARMCLOCK Not yet provided timer interrupt with priority 1.
4	OUT-OF-STACK= (LAMBDA()(S,ERRORLOOP1 17 "STACK-FULL")) has very high priority 5.
5	OUT-OF-HEAP = (LAMBDA()(S,ERRORLOOP1 18 "HEAP-FULL")) has very high priority 5.
6	RECLAIM modest priority 3.
7	Currently, UNUSED-CHANNEL7 with priority 1.
8...n	Currently unallocated.

TRAPS

Traps are program, or endogenous, events that happen synchronously. Like external events they are divided into classes and each program event is associated with a program events service channel. Unlike external events they may receive operands, and may return a value.

A principal use for program events is error handling. Errors are detected and various program event channels are used to provide error servicing. Several classes of errors occur in LISP:

1. LISP machine check --- The LISP state is not recoverable and the error is uncorrectable. The only user actions possible correspond to debugging in the micro-code (with respect to the fiction of there being a LISP machine), stopping or abnormal termination, and resetting or restart. No user service channels are provided for errors of this class.

2. Uncorrectable error --- The LISP state is well defined, but there is no meaningful recovery. In such cases user channels are invoked but if the channel attempts to return a value an automatic unwind occurs.

3. Correctable error --- The LISP state is clean and it is possible to proceed if the user service channel provides a value.

See Table 2, for more explicit details for each channel. There are distinctions that are not a property of the service expression but rather how it is invoked. Consult the Program Description and Operations Manual (SH20-2076-0) and listings for details about ERROR, ERRORR, ERR2, ERR4 and such if you become mystified.

Most program events run by causing the n^{th} element of PROGRAM-EVENTS to be applied to n, the arguments and the current state. The current implementation of most of the error channels is to call S,ERRORLOOP, the break state supervisor.

Table 2. The Program Events Channels

Ch. No.	Purpose, Explanation, and Initial value	Value of ?ARGS?	Value Expected
1	No current purpose. Initial value NIL.	n.a.	n.a.
2	"FR DOMAIN ERROR" Initially: S,ERRORLOOP	$(a_1 \ldots a_n$ *ur*)	*e* which user supplies for reevaluation.
3	"NON-CONFORMAL MACRO APP " Expect user to supply expression for correct value of the macro application, or to (UNWIND). Initially: S,ERRORLOOP	((*rator rand*...) x) where x is macro being applied.	*e* which user supplies for reevaluation as the value of the macro application.
4	"NON-CONFORMAL APP " Expect user to supply expression for correct value of the application, or to (UNWIND). Initially: S,ERRORLOOP	$(a_1 \ldots a_n$ x) where x is function being applied.	*e* which user supplies for reevaluation.
5	"DYNAMIC MACROS NOT ALLOWED " Attempted to apply a macro to computed operands. Expect user to supply expression for correct value of the application, or to (UNWIND). Initially: S,ERRORLOOP	$(a_1 \ldots a_n$ x) where x is macro being applied.	*e* which user supplies for reevaluation.
6	"APP OF THE INAPPLICABLE " Application of a constant or expression that evaluates to itself. Usually means undefined (i.e. inapplicable) function. Expect user to supply expression for correct value of the application, or to (UNWIND). Initially: S,ERRORLOOP	$(a_1 \ldots a_n$ x) where x is constant being applied.	*e* which user supplies for reevaluation.
7	"NON-SD 2ND ARG " <u>EVAL</u> or <u>CLOSURE</u> was not given an sd as second argument. Expect user to supply expression for correct value of the evaluation, or to (UNWIND). Initially: S,ERRORLOOP	(*e* y ... <u>EVAL</u>) where y was supposed to be a *sd*.	*e* which user supplies for reevaluation.
8	"ARITHMETIC ROUTINE ERROR" Expect user to supply expression for correct value of the evaluation, or to (UNWIND). Initially: S,ERRORLOOP	$(a_1 \ldots a_n$ x) where x is routine being applied.	*e* which user supplies for reevaluation.

9	"OUT OF STATEMENT CONTEXT GO " GO expression occurred out of statement context. Expect user to supply expression for correct value of the evaluation, or to (UNWIND). Initially: S,ERRORLOOP	(GO *st-lab*)	*e* which user supplies for reevaluation. (probably not dynamically correctable)
10	"NO SUCH LABEL TO GO TO " (GO *st-lab*) in statement context has no corresponding *label*. User is placed in break loop but control will not return to the offending statement context, instead an UNWIND will occur. Initially: S,ERRORLOOP	(GO *st-lab*)	n.a.
11	"1ST ARG TO SET NOT ID " Attempted assignment to a non-*id*. Expect user to supply expression for correct value of the evaluation, or to (UNWIND). Initially: S,ERRORLOOP	(y x SET) where y is not an *id*.	*e* which user supplies for reevaluation.
12	"USER CALLED ERROR W/ RETURN EXPECTED " The explicit call to ERROR channel. The argument is provided in the expression (ERROR mes). Expect user to supply expression for correct value of the evaluation, or to (UNWIND). Initially: S,ERRORLOOP	*s-exp*	*e* which user supplies for reevaluation.
13	"NON-CONFORMAL LABEL-EXP " Non-conformal *label-exp*. Expect user to supply expression for correct value of the evaluation, or to (UNWIND). Initially: S,ERRORLOOP	*label-exp*	*e* which user supplies for reevaluation.
14	"USER CALLED ERROR W/ UNWIND EXPECTED " ERROR with explicit unwind provided. Expect user to look around at his state. Initially: S,ERRORLOOP	*s-exp*	n.a.

Summary and Future Directions

This document should have supplied some of the necessary information that the PDOM neglected. It will not serve as a good primer for to LISP370. Little attempt was made to justify the choices made in the design. It describes the available LISP/370 system and not our experimental design studies. Those topics must be left to some future papers.

Acknowledgements

The author has benefited greatly from the conversations and contributions of Arthur C. Norman, Martin Mikelsons, Allen L. Brown, Vincent J. Kruskal, Jean Rivlin, Richard W. Ryniker, Bill Burge, Andrew P. Black, and Cyril N. Alberga who was my co-worker and sharpest foil.

965 - IBM - 04

BIBLIOGRAPHY

There are a number of surveys and reports justifying LISP. The following may be useful:

- John McCarthy, "Recursive Functions of Symbolic Expressions and their Computation by Machine," *Communications of the Association for Computing Machinery*, vol 3, 1960, and "A Basis for a Mathematical Theory of Computation," in *Proceedings of Western Joint Computer Conference*. 1961. The original papers on LISP.

- D. G. Bobrow, and B. Raphael, "New Programming Languages for AI Research," *Association for Computing Machinery Computing Surveys*, vol. 6 no. 3, September 1974.

- Carl Engelman, *Engineering of Quality Software Systems (Towards an analysis of the LISP Programming Language)*. NTIS Report no. RADC-TR-74-325, Volume IV. U. S. Govt. Accession No. AD/A-007 769, 1975.

There are a number of LISP textbooks. The following, in particular, may be of interest:

- C. Weissman, *LISP 1.5 Primer*, Dickenson Publishing Company, Belmont, California, 1967.

- Laurent Siklossy, *Let's Talk LISP*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

- John R. Allen, *Anatomy of LISP*, McGraw-Hill, New York, 1978. An outstanding new book.

- W. D. Maurer, *A Programmer's Introduction to LISP*, American-Elsevier Publishing Company, Inc., New York, 1973.

- D. P. Friedman, *The Little LISPer*, Science Research Associates, Inc., Palo Alto, Calif., 1974. A programmed text approach to teaching elementary recursion.

LISP programmers manuals:

- John McCarthy *et al*, *LISP 1.5 Programmer's Manual*, M.I.T. Press, Cambridge, Massachusetts, 1965.

- David A. Moon, *MACLISP Reference Manual*, Project MAC, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1974.

- Warren Teitelman, *INTERLISP Reference Manual,* available from Xerox Palo Alto Research Center, Palo Alto, California, or Bolt, Beranek and Newman, Cambridge, Massachusetts, 1974.

LISP/370 Documents available through your local IBM Branch Office:

- "Availability Notice", Program Number 5796-PKL.
- "Program Description/Operations Manual", SH20-2076-0.
- "Marketing Guide", Number ZZ 20-4287, IBM INTERNAL USE ONLY.
- Jon L. White, *LISP/370: A Short Technical Description of the Implementation.* Submitted to SIGSAM Bulletin, also included in "Marketing Guide Number".

AI general:

- Patrick Henry Winston, *ARTIFICIAL INTELLIGENCE*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1977. A source book on AI, with chapters on LISP.