THE DEFINITION OF LISP1.8+0.3i

F. W. Blair
IBM Thomas J. Watson Research Center
Yorktown Heights, N. Y. 10598

Note: This is a DRAFT document. Do not believe it (yet). Furthermore work has been suspended (if not moribund) since 1979. As a courtesy to this living hacker, please do not distribute.

ABSTRACT

This definition document serves as the specifications for a LISP system that is under development at the IBM T J Watson Research Center. It is an abstract description dealing with the specification of: Syntax, Semantics, Machine States, Data Objects, and Primitive Operators. It attempts to capture, in some detail, an evolved and still evolving design with particular attention to pragmatics.

IBM Internal Use Only

CONTENTS

IBM Internal Use Only

IBM Internal Use Only

IBM Internal Use Only

INTRODUCTION

The LISP1.5 system[8] has been and continues to be a means for the study and development of programming science. It provides an evaluation model which explains many notions common to programming technology. LISP provides data objects and primitive operators which shape the universe of discourse for LISP programmers in much the same way that our natural language lexicons shape or limit our thoughts. LISP systems also usually contain that ultimate admission of their own incompleteness: namely the flexibility to be extended and modified.

Little attempt at either intuitive or denotational semantics is made in this paper; it merely posits the computational or operational semantics of a dialect of LISP which is thought to be representative of current practice. The intent is to move the debate on LISP fundamentals into sharper focus and to encourage public review. Mainly it serves as the definition for an experimental system, currently under development, designated "LISP1.8+0.3i".

The family of languages designed toward the goal of a simple, formal definition of the basic characteristics of programming languages, and based on function application has been called the applicative programming languages. LISP will be used to designate that subset which has some form of *s-expression* data language which also is used as the expression language. The LISP1.x varieties are those which bear striking similarities to LISP1.5.

The reader should refer to Reynolds[10] for a systematic review of definitional interpreters. That paper contains much motivational and descriptive discussion about language classification and language features. The informal discussion at the beginning of his paper is relevant to this paper. The reader should note the similarity between Reynold's continuations and the state descriptors of this paper.

Of theoretic interest to the subject of LISP is Gordon's thesis[5] on "Models of Pure LISP", in which he presents a Scott[11] style of denotational semantics for pure LISP (chapter 1 of McCarthy[8]) as well as an operational semantics schema and a proof of their equivalence. More recently Newey's thesis [9], "Formal Semantics of LISP with Application to Program Correctness" is recommended.

In these works the authors were primarily concerned with descriptions which enable and encourage proof. The emphasis in this paper is on description of the underlying processor. The attempt here is to expose certain pragmatically significant points.

The works of Steele and Sussman [12, 13, 14] are particularly relevant to this work. They cover much the same ground, namely definition of LISP, and they deal in considerable detail with the pragmatics. A fundamental difference between their work and this is that they have emphasized the static determination of programs, while this work retains much of the dynamic evaluation capabilities of LISP.

In this model of LISP considerable emphasis is placed on the effect of an operator on the machine state. Contexts and their manipulation are also emphasized. It is thus a computational semantics rather than a denotational semantics. The GO expression which might otherwise have been derived is featured because of its unique characteristic of not requiring a new state and only affecting the control and stack of the current state. The introduction of processor concerns has made the model for LISP more complex due to the attempt to describe more phenomena; and because of that complexity subtle errors may have crept into the elaborate machinery this paper attempts to describe. It appears however, that for a modest expenditure in metalinguistic cogs and wheels, considerable descriptive power is achieved. Much of the phenomena of programming is resolved by this mode of description into discrete mechanisms.

LISP1.8+0.3i is a language of expressions (*e*) . These expressions are a subset of a data language for LISP called symbolic expressions (*s-expression*). The concrete canonical form for external representation of LISP1.8+0.3i expressions is practically devoid of syntactic niceties. Normally these niceties are present to aid in the human recognition process. LISP expressions are abnormal in this respect and for good and sufficient reasons. The syntax of expressions can be thought of as one which reveals the simplicity of the underlying abstract syntax. This simplifies the recognition processes of the READ function and other such processes that examine the language representation. The so called ugly canonical form representation choice does not exclude alternative representations which would be more palatable, it is merely the didactic choice.

Much of the art that has been created in the LISP community serves to enrich the basic LISP systems and to bring joy to their users. This paper will eschew such user orientated delights and focus on rather more mundane system programming features. Its main purpose will be to try to develop a reasonably abstract but workable model of the computationally interesting problems of a "LISP machine".

The computational semantics of LISP is the relation of an expression to the data value it denotes, the intermediate states produced, and the state of the machine that obtains after the denotation was produced. Intuitively, the semantics may be viewed as the process (called evaluation or interpretation) to which expressions are subjected to produce values which they are said to denote. Expressions (which are like the phrases of natural languages) are very often replete with a form of pronominal reference called a variable. Such constituent expressions can only have meaning with respect to a context (called the environment) which gives the meaning of such variables.

In the past, the semantics of LISP has been given by the process of self description. Perhaps this stems from a desire to illustrate the power of the LISP language, but more pragmatically it results from the method used in a "bootstrap" implementation of LISP. Needless to say, this approach has some shortcomings from the point of view of definition. A tacit understanding of LISP is required to read the definition of LISP and at least a primitive LISP system is required to begin the bootstrap. This model will deviate from this tradition.

NOTATIONAL CONVENTIONS

The following conventions are meant to be helpful and are explicitly described for future reference. It must be admitted that there is rather more than one would like to instantly commit to memory. The reader might find it advisable to scan briefly and then refer back as needed.

{ and } are used for metalinguistic grouping.
[and] are used to indicate optionality.
$^+$ is used to indicate one or more.
| is used to separate alternatives.
Vertical alignment is also used for alternatives.
The ellipsis "..." is used to denote zero or more objects. Thus x... means zero or more x's, but ...x means zero or more of anything but x and then x.

Subscripts will be used to indicate a required one-to-one correspondence, whenever the intent is not clear. They will also serve to denote individual members (not necessarily identical) of a class.

Identifiers (names) given entirely in upper-case letters are LISP1.8+0.3i ordinary identifier data objects, they are used as variables and statement labels in the LISP language. Lower-case identifiers are used as metalinguistic variables ranging over LISP data objects. The normal font is also used in the semantics rules for commentary and as logical metalanguage.

Italics will pertain to syntacticly defined objects (also abstract syntax objects). Metalinguistic variables ranging over a syntactic class of LISP expressions or data objects are represented by the name of the class in lower-case italic. Syntactic classes are represented by the class name in upper-case italic.

Boldtype will pertain to the metalinguistic state. The following are constants of the metalanguage: ***PRED***, ***ES***, ***OP***, ***LABEL***, ***SF***, ***MAPP***, and ***EVAL***. $\boldsymbol{APP_1}$, $\boldsymbol{APP_2}$, $\boldsymbol{SEQ_{\{1|2\}}}$, ***MU*** and ***REAP*** form composite meta-symbols which have ***S*** as a component. These meta-symbols are truly the cogs and wheels of the meta-language. Each performs a single function which may be of interest to the implementer.

Bold, upper-case italic letters will also be used to designate the metalinguistic state components. It is hoped that the use of bold italic fonts for the metalinguistic domain will be helpful.

'(' , ')' , '•' , '=' , '%', and blanks are all used as special symbols in forming *s-expression* representations (also called *s-exp* or *datum*).

";" is used as a metalinguistic separator.

"●" is used as a metalinguistic infix CONS operator which associates to the right. When used as a prefix it is the identity operator. "●" differs from • the LISP basic operator

CONS but is similar in concept. "*(*" and "*)*" are used to form metalinguistic list models; they are not necessarily LISP data objects.

LISP OBJECTS

There are several domains of discourse that pertain to LISP programming. There is the blackboard, or external representation as characters, domain of LISP data objects. A canonical concrete syntax for these objects is familiar to LISP programmers but is not unique. Other external representations are possible. There is the data processor's domain of LISP data objects in a memory. There is the evaluator's domain of LISP states and data objects in a memory. In LISP it is not uncommon to pretend that these domains are isomorphic. This document maintains such a pretense on the grounds that the familiar external notation (augmented as needed) will suffice as the abstract syntax. The reader must decide from context which domain pertains. Usually it is the domain of LISP states and data objects in a memory. Thus, the internal memory domain objects are denoted by external domain representations.

An *s-exp* is:

[*label*]{*c* | *id* | *funarg* | *sd* | *combination* }
where *label* is {*label-name* = }, and
where *label-name* is {%L*digit*$_1$...*digit*$_n$} where $1 \leq n \leq 8$ and,
where *id* ϵ *ID* the set of identifiers (names), and
where *c* ϵ *C* the set of constants, and
where *sd* ϵ *SD* the set of state descriptors for which no written representation is intended.
An *sd* has a LISP machine state {*S;E;C;D*} as a component. (See below.)
where *funarg* = %(FUNARG *e* • *sd*)
where *combination* = (*comp*$^+$ [• *comp*])
where *comp* is {*label-name* | *c* | *id* | *funarg* | *combination* |
{*label comp*} }

It should be noted that the data language of LISP1.8+0.3i *s-exp's* is somewhat richer than is given above. For example the set *C* may include selector structures and vectors. A more extensive syntax of *s-exp* will be given elsewhere. The syntax as given is sufficient for the representation of LISP1.8+0.3i and for this explanation. It should be noted that *s-exp*'s are allowed to have themselves as components. A common practice in LISP systems is to provide READ and PRINT functions which preserve EQUAL-ity for all of the above except *sd*, and EQ-uality for all *id* except a special subclass called *gensyms*.

The distinguished constant *nil* is written () and is included in *C* . The following denote constants known to the evaluator: LAMBDA, MLAMBDA, MU, QUOTE, FUNCTION, SETQ, LABEL, COND, SEQ, GO, EXIT, PROGN, RETURN, FR*CODE, AUX, SETX.

These data constants are not to be confused with the identifiers. The ultimate external representation of these constants is defined elsewhere!

THE METALINGUISTIC MACHINE

I shall copy the method of P. Landin[5] in creating a metalinguistic description of a machine. The machine itself is described as a complex space consisting of "states" and the state transition functions. The meaning of an expression is given by including it in an initial state of this machine, and when a terminal state is reached after repeated transitions the meaning of the expression may be extracted. The states of the machine are quadruples {***S;E;C;D***} whose components are called Stack, Environment, Control and Dump, respectively. The term activation-record is commonly used for such an entity. This state language will be used to give meaning to the expressions of LISP1.8+0.3i. Revealing the state components seems to simplify the description of certain concepts. The state language is also suggestive of implementation strategies.

The state components are:

S = The value Stack, modeled herein as a metalinguistic list of *s-exp*'s.

> The Stack provides temporary storage for computed values and its usage gives rise to transmission conventions for the passing of parameters.

E = The Environment modeled herein as a metalinguistic list structure.

> The purpose of the Environment is to determine the value of a variable. The Environment is said to provide a context with respect to which an expression is said to have a value. A somewhat more elaborate view is that the Environment provides the storage spaces that contain the values denoted by variables. ***E*** may be viewed as an object having the following structure:
>
> ***E*** is either ***nilE*** or ***(lE • tE)***,
>   where ***tE***, the tail of environment, is an ***E***,
>   and ***lE***, the lexical environment, is ***(contour • tlE)***,
>   where ***tlE***, the tail of the lexical environment, is *()* or an ***lE***,
>   and ***contour***, the head of the lexical environment, is ***(binding ...)***,
>   where ***binding***, is ***(value • ident)***,
>   where ***value*** is an *s-exp*,
>   and ***ident*** is {*c* | *(*%=FLUID • ***iden)*** | *(*%=LEX • ***iden)*** | ***iden***}.
>   where ***iden*** is { *(*[%=*type-name*] *id)* },

As a matter of convenience ***ehE = ()*** is used to indicate an empty contour. The empty environment ***nilE*** is ***((ehE)•())***.

It may be helpful to informally explain this structure description. Firstly it should be remembered that this is a metalinguistic model and not a data list structure. Note that an ***E*** typically contains a reference to another ***E***. This is the "inherited context" and will be detailed latter. It has a lexical part consisting of a list of ***contours*** each of which is a list of ***bindings*** and each ***binding*** has a place for a ***value*** and associated information as to the name, type, and availability of that binding. It will be seen that only the bindings designated FLUID are available in ***tE***, the inherited context or tail of environment.

E is constructed from a bound variables template *bv*, the argument parameter *arg* and another Environment ***E*** by a three-place constructor:

```
bind{bv; arg; E } where
        bv is { ident | ($bv_1$ • $bv_2$) }
        arg is { atom | ($arg_1$ • $arg_2$) }
            where atom is { id | c } .
        and ident is { c | (%=FLUID • iden) | (%=LEX • iden) | iden},
            where iden is { ([%=type-name] id) },
```

```
bind{bv; arg; E } =
      if bv is a c then E,
      if bv is an ident then
            (((( conform{arg;ident} • ident) • contour) • tlE) • tE) ,
               where conform{arg;ident} =
                     %=type-name{arg} if ident contains %=type-name else arg,
                     where %=type-name{arg} = arg if it is of the correct type other-
                     wise it gives a domain error.
      if bv = ($bv_1$ • $bv_2$) and arg is atom then
            bind{$bv_2$; er{$bv_2$}; bind{$bv_1$; er{$bv_1$};E}}.
      if bv = ($bv_1$ • $bv_2$) and arg is ($arg_1$ • $arg_2$) then
            bind{$bv_2$ ; $arg_2$ ; bind{$bv_1$ ; $arg_1$ ; E}}.
            There are no other cases.
```

er{x} = () if x is a *c*, **er**{x} = x otherwise. The otherwise case is called the non-conformal arguments case.

bind$_2$ will be used when *bv* is bound to multiple arguments. Notice the possible construction of a list of trailing operands.

```
bind₂{bv;E; $a_1$;...$a_n$} where $0 \le n \le 255$ is:
      if bv is a c then E and,
      if n is 0 i.e., no arguments then bind{bv;();E} and,
      if bv=($bv_1$•$bv_2$) and $a_1$ exists, then bind{$bv_1$; $a_1$; bind₂{$bv_2$;E; $a_2$;...}} and,
      if bv is an id then bind{bv;($a_1$...$a_n$);E}.
```

E is described above as a metalinguistic data structure. In what follows that structure will be viewed as defining a function. *E* as described below is actually a combination of three metalinguistic functions on variables, namely: **lookup**, **assignment**, and **type-recall**. These are combined to show their similarity and mutual dependence on the structure of *E*. As described below, the current lexical environment contours are searched for both lexical and fluid bindings of *id*. In the tail of the environment (i.e. that which is not local) only fluids are sought. In the case of assignment the value being assigned must conform to the type associated with the binding.

```
$E\{id_l;x;y\}$ =
    if $E = nilE$ then some agreed upon global binding, global{$id_l$;x;y},
    if $lE$ the lexical environment is empty, i.e. $()$ then fluid{$id_l$;x;y;$tE$},
    if $lE = (() \bullet tlE)$, i.e. the contour is empty, then $(tlE \bullet tE)\{id_l;x;y\}$,
    if $contour = (binding_l \bullet contour_l)$ and id-of{$binding_l$} $\neq id_l$ then
        $((contour_l \bullet tlE) \bullet tE)\{id_l;x;y\}$ ,
    if $contour = (binding_l \bullet contour_l)$
        where $binding_l = (s\text{-}exp_l \bullet ident_l)_l$ and id-of{$binding_l$} $= id_l$ then
            if x = LOOKUP then (conform{$s\text{-}exp_l$;$ident_l$} $\cdot$ $ident_l$)$_l$
                Comment: Conformation in the case of lookup only seems
                redundant. This will provide facility for monitoring the use
                of a given binding. Problem: How does the underlying
                %=type-name function know if it is being used for the
                LOOKUP or the ASSIGN option? The obvious answer to
                try is to explicitly pass these as parameters to conform and
                %=type-name.
            if x = ASSIGN then (store{conform{y;$ident_l$};$binding_l$} $\cdot$ $ident_l$)$_l$
            if x = TYPE then type-of{$ident_l$},
    There are no other cases.
```

fluid{*id*$_l$;x;y;*E*} is similar to *E*{*id*$_l$;x;y} except that **fluid-of** is used instead of **id-of**, i.e., it matches only those bindings which were explicitly mentioned as (%=FLUID · *iden*) at bind time. (For more details see section describing: Global Environments, Exit Routines, and An Efficiency Device.)

fluid-of, **id-of**, and **type-of** are obvious selector functions.

The function **store** requires the concept of a memory as an addition to the model.

All the state transformations given below are of the form:

$\{S; E; C; D\} \rightarrow \{S_2; E_2; C_2; D_2\}$

and will be thought of as having taken place in a memory *M*.

$\{S; E; C; D\}M \rightarrow \{S_2; E_2; C_2; D_2\}M_2$

These memory concepts are described in Bekic' and Walk[3], and also Reynolds[10].

((ehE) • *E)* is an example of an environment in which only the fluid and global bindings are accessible.

global{*id$_I$*;x;y} = ***gloE{id$_I$;x;y}***

A ***gloE*** is a special object with two components:

glonot the binding-not-present prescription for this ***gloE***
is a pair (*gloval* • *gloalo*) where
gloval is NIL or else a two argument function
from the *id* in question 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.

and *glolst* the global data list structure environment is
({*glodat* | *globnd*} • {*glolst* | *glotrm*}),
and *globnd* the global binding is a *pair* (*s-exp* • *ident*),
and *glodat* the global own data, is any *s-exp* which is not a *pair*,
and *glotrm* the global environment terminator is, {NIL | *sd*}.

(For more details see section describing: Global Environments, Exit Routines, and An Efficiency Device.)

nilE the distinguished empty environment acts as a terminator for that part of the environment created by **bind**, which shall be referred to as the normal environment.

The following definition of the global environment function provides capabilities that could be used for the production of bindings on first reference, global context switching, direct access data bases, and in general is limited only by the imagination of the programmer.

```
$\textbf{\textit{gloE}}\{id_l;x;y\}$ =
if $globnd_l$ = lookup{$id_l$;$glolst$} is ($s\text{-}exp_l$ • $ident_l$)$_l$ then
        if x is LOOKUP then $globnd_l$
        if x = ASSIGN then ( store{conform{y;$ident_l$};$globnd_l$} • $ident_l$)$_l$
        if x = TYPE then type-of{$ident_l$},
if $globnd_l$ = (), it was not found, then
        if $gloval$ is (), and $gloalo$ is (), then
                if x is LOOKUP, the lookup default pertains, then
                        rplac-glolst{$binding_l$=($id_l$ • $id_l$)}
                                where rplac-glolst returns $binding_l$ after replacing the $glolst$
                                of $\textbf{\textit{gloE}}$ with its argument.
                if x is ASSIGN, the assignment default pertains, then
                        rplac-glolst{(y • $id_l$)}
                if x is TYPE, then
                        type-of{rplac-glolst{ ($id_l$ • $id_l$)}}
if $gloval$ is = (), and $gloalo$ ≠ (), then $gloalo$\{$id_l$;x;y;$glolst$\}.
if $gloval$ is ≠ (), and $gloalo$ = (), then rplac-glolst{($gloval$\{$id_l$;x;y\} • $id_l$)}
else $gloalo$\{$id_l$;x;$gloval$\{$id_l$;x;y\};$glolst$\}.
```

C = The Control stack, is homologous to a list of instructions and data, it has two possible forms:

1. In the case of LISP1.8+0.3i interpretive execution, *C* is modeled herein as a metalinguistic list of *s-exp* and meta-symbols.
2. In the case of machine execution, $\textbf{\textit{lc}}_{name}\{bpi\}$ denotes some location in the binary program image *bpi* which *name* characterizes.

The nature of *C* is sufficient to characterize these two modes of execution. Throughout these descriptions Stack and Control are represented as independent structures. The actual use of them indicates a preference for a dependence of *S* on *C*. Perhaps it is best put this way, whenever the *C* of a state is the same the number of elements on *S* is the same.

D = The Dump, i.e. a reference to a previous state,
which is either *()* or a previous state $\{S_2 ; E_2 ; C_2 ; D_2\}$.

Certain distinguished states will be denoted by subscripting *D* with an identifier. For example, $D_{non\text{-}conformal\text{-}app}$ denotes a state whose description is given in Appendix A.

Because *D* defines the chain of states from which control descended, it is sometimes called the control-chain. It is possible to define a control-chain environment.
If we consider a state $D = \{S_2 ; E_2 ; C_2 ; D_2\}$,
where lE_2 is the first lexical environment component, we can view this as a list of lexical environments where D_2 gives the rest of the list.

We can then define the operator **dynamic** which is analogous to the function ***E*** except that its analogue to **fluid** uses the ***D-part*** of a state rather than the ***tE*** of an ***E***.

The metalinguistic list models are meant to be suggestive, there is no decree that ***S***,***E***,***C***, or ***D*** must be a LISP data-object of type "list" residing in the heap. It is in fact the case that the system being developed uses a retention stack model similar to that described by Bobrow and Wegbreit[4]. A certain amount of reader good will is required here as explicit list operators will sometimes be used. Meta-linguistic "CONSing" will be indicated as "●" and ordinary "CONSing" by "•".

EXPRESSIONS

A LISP expression *e* is one of:

c a constant where (), *bpi*, *sd*, *sf*, *ur*, *abstraction*, *constant-closure*. $\in C$

where *bpi* is a compiled program value object which is:

an *mbpi* a compiled macro

or an *fbpi* a compiled function

where *sd* is a state descriptor

where *sf* is $\in$

{ LAMBDA | MLAMBDA | MU | QUOTE | SETQ | FUNCTION | LABEL | COND | SEQ | GO | EXIT | PROGN | RETURN | FR*CODE | AUX | SETX}

where *ur* is an understood primitive operator which is:

a *fix-ur* for primitive operators that require a definite number of arguments or a *mult-ur* which is an understood operator that takes an indefinite number of arguments

where *abstraction*, is either:

a *lambda-abstraction* %(LAMBDA *bv* • *exp-seq*)

where *bv* the bound-variable part is { *ident* | (bv_1 • bv_2) }

and *ident* is { *c* | (%=FLUID • *iden*) | (%=LEX • *iden*) | *iden*}

where *iden* is { ([%=*type-name*] *id*) }

and *exp-seq* is {*atom* | (*e*...)}

or an *mlambda-abstraction* %(MLAMBDA *bv* • *exp-seq*)

or an *mu-abstraction* %(MU *bv* • *valuelist*)

and *valuelist* the values list is (*s-exp*...)

or a *sequence-abstraction* %(SEQ *tag aux s*...)

the sequence label *tag* is an *id* or ()

the auxiliary-stack-place names list *aux* is (*iden* ...)

each statement *s* is a:

statement label *st-lab* which is an *id*, or

program-statement *ps* which is an *e* which is not an *id*

or a *operator-code-abstraction* %(FR*CODE *e f-list* • *lap-code*)

where *f-list* and *lap-code* are described in the LISP assembler documentation

where a *constant-closure* %(CCLOSE *e-part* • *s-part*)

where the expression part *e-part* is an *e*

and the state part *s-part* is an *sd*

id a variable

funarg a closed expression %(FUNARG *e-part* • *s-part*)

where the expression part *e-part* is an *e*

and the state part *s-part* is an *sd*

(*rator* • *randlist*) a *combination*

where the operator *rator* is an *e*

and *randlist* the operands list is (*rand*...)

and each operand *rand* is an *e*

Informally the evaluation of constants is rather simple: they evaluate to themselves. Certain classes of constants may be applied as operators. There are applicative constants and what is meant by their application is described in detail in the following sections.

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 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 (namely state descriptors) 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 appropriate 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 <u>fluid</u> <u>binding</u> is accessible to any evaluation of a variable for which it is the most recent binding in the inherited environment. A <u>lexical</u> <u>binding</u> is not accessible to CALLed or non-lexical operator expressions and thus offers some degree of isolation from side effects. The accessibility of lexical variables is an important concern for the semantic rules that follow.

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 but 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.

The funarg construct is an expression which contains an expression-part and a state. The value of the funarg is the value of its expression-part evaluated with respect to the environment of its state.

It should be noted that except for constants, funargs, and variables; every LISP expression is a combination. Some of these combinations are distinguished for semantic reasons. The combination form is used to indicate application. There are three types of application expressions:

1. Meta combinations, a transformation from an operator value which is special form applicable and the unevaluated list of 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-abstraction*, or a closure (*macro-funarg*) 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. 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.

The following constants (*sf*) occur as *rator* value and denote special forms, i.e. their application is special and defined by special rules.

{ <u>LAMBDA</u> | <u>MLAMBDA</u> | <u>MU</u> | <u>QUOTE</u> | <u>SETQ</u> | <u>FUNCTION</u> | <u>LABEL</u> | <u>COND</u> | <u>SEQ</u> | <u>GO</u> | <u>EXIT</u> | <u>PROGN</u> | <u>RETURN</u> | <u>FR*CODE</u> | <u>AUX</u> | <u>SETX</u>}

mu-abstractions and *funargs* whose *e-part* are *mu-abstractions* or *sf's* also apply specially.

As such special forms apply specially i.e., they are applied to their unevaluated *randlist*, they often require that *randlist* have a definite syntax. The required syntax for these built-in operators is illustrated below:

An *abstraction-exp*, which is either

a *lambda-exp* (e_1 *bv* • *exp-seq*)

where e_1 has the value LAMBDA.

and *bv* the bound-variable part is { *ident* | (bv_1 • bv_2) }

and *exp-seq* is {*atom* | (*e*...)}

or an *mlambda-exp* (e_1 *bv* • *exp-seq*).

where e_1 has the value MLAMBDA.

or an *mu-exp* (e_1 *bv* • *randlist*).

where e_1 has the value MU.

or an *operator-code-exp* (e_1 e_2 *f-list* • *lap-code*).

where e_1 has the value FR*CODE.

or an *sequence-exp* (e_1 *tag aux s*...).

where e_1 has the value SEQ.

(e_1 *s-exp* • *s-exp*) where e_1 has the value QUOTE, is a quoted s-expression.

(e_1 *id e*) where e_1 has the value SETQ, is an explicit assignment.

(e_1 *id e*) where e_1 has the value SETX, is an auxiliary-stack-place assignment.

(e_1 *id*) where e_1 has the value AUX, is an auxiliary-stack-place contents fetch.

(e_1 *bv* e_2 • *s-exp*) where the value of e_1 is LABEL is a label-expression.

(e_1 {*c* | *id* | (*p* •[*q*])}...) where the value of e_1 is COND is a conditional-expression.

where the predicate *p* is an *e*, and

the consequent *q* is an *exp-seq*.

(e_1 • *exp-seq*) where the value of e_1 is PROGN is an expression sequence

(e_1 *tag aux s*...) where the value of e_1 is SEQ is a statement sequence expression

(e_1 *st-lab* • *tag*) where the value of e_1 is GO is a go-expression.

(e_1 { *id* | *ps* } • *tag*) where the value of e_1 is EXIT is an exit-expression.

(e_1 e_2 • *s-exp*) where the value of e_1 is RETURN is a return expression.

(e_1 e_2 • *s-exp*) where the value of e_1 is FUNCTION is a closure expression.

The following basic operator constants (*ur*) occur as *rator* values, they are ordinary applications which are defined by special rules:

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

Other basic operators, such as those defining data primitives, are presumed but not defined at this time.

EVALUATION

The following is a listing of the state transitions for the {*S*; *E*; *C*; *D*} machine. The evaluation of *e* with respect to *E* is given by {*()*; *E*; *e•()*; *()*}. Throughout the rules that follow the most important determinate of what happens next is the object at the head of *C*.

Simple State Transitions

1. Halting

$\{x \bullet S;\ E;\ ();\ ()\}$ $\rightarrow$ HALT, the result of the computation is x.

Comment: The control which is a stack of expressions awaiting evaluation is empty and there is no previous state to restore. In practice halting never occurs as control returns to some operating system. This rule is here for theoretical completeness and is not one of the usual transition states.

2. Value return restoring the former state

$\{x \bullet S;\ E;\ ();\ \{S_2;\ E_2;\ C_2;\ D_2\}\} \rightarrow \{x \bullet S_2;\ E_2;\ C_2;\ D_2\}$

Comment: This rule models single-valued procedure-exit. The control of this state is empty and the dump is not empty, therefore the last value computed (the one at the head of the stack) is returned to the former state.

3. Re-evaluation

$\{x \bullet S;\ E;\ EVAL \bullet C;\ D\} \rightarrow \{S;\ E;\ x \bullet C;\ D\}$

Comment: The meta-symbol ***EVAL*** is used to indicate reevaluation after macro expansion. Used in rules 8.1.1.2.1 and 8.2.

Constants

4. Self-denoting expression

$\{S;\ E;\ c \bullet C;\ D\} \rightarrow \{c \bullet S;\ E;\ C;\ D\}$

Comment: Constant expressions are idempotent, that is, they evaluate to themselves. All data objects other than pairs, funargs and identifiers are idempotent under this rule.

Variables

5. Evaluation of a variable

$\{S;\ E;\ id \bullet C;\ D\} \rightarrow \{\textbf{contents}\{E\{id\}\} \bullet S;\ E;\ C;\ D\}$.
where **contents**{(val ...)} = val.

Comment: Here we see the use of the environment *E*, it gives meaning to variables. The mechanism for user installed global interpretations was discussed earlier.

Closures

6. Closure evaluation

$\{S;E;funarg \bullet C;D\}$ and *e* is the *e-part* and *sd* is the *s-part*.
$\rightarrow \{();\ ((ehE \bullet lE_l) \bullet tE_l);\ e \bullet ();\{S;\ E;\ C;\ D\}\}$ where *sd* has E_l as its ***E-part***.

Comment: This expression represents an expression closed with respect to the environment of *sd*. In the absence of updating, or state saving, such expressions denote the same value regardless of the context in which the closure is evaluated. Abstractly it would be sufficient to reference *E* but in LISP1.8+0.3i it is achieved as described. *funargs* are to be contrasted with *constant-closures*, they both APPLY the same but the later is self-denoting whereas the former may denote expressions that require further evaluation. Another nuance of LISP1.8+0.3i is that along with the new state there was created a new lexical environment with an empty first contour and a reference to the lexical environment of the *sd* as its tail.

Combinations or Application Expressions

7. Operator evaluation

$\{S;\ E;\ (e_l \cdot randlist) \bullet C;\ D\} \rightarrow \{S;\ E;\ e_l \bullet OP \bullet (e_l \cdot randlist) \bullet C;\ D\}$

Comment: In this common case the operator expression is first evaluated for classification.

8. Operator value determines what happens next

$\{x \bullet S;\ E;\ OP \bullet (y \cdot randlist_l) \bullet C_l;\ D\}$

8.1. Understood special form if x is {*sf* | %(<u>FUNARG</u> *sf* • *sd*)}.

8.1.1. Operator tested for lexical application and closure avoidance.
If $C_1 = OP \bullet (\mathrm{w} \cdot randlist_2) \bullet C_2$

8.1.1.1. Ordinary application with explicit *lambda-exp* as operator.
$\{\underline{\mathrm{LAMBDA}} \bullet S_1;\ E;\ OP \bullet (\mathrm{y} \cdot (bv \cdot exp\text{-}seq)) \bullet\ OP \bullet (\mathrm{w} \cdot randlist_2) \bullet C_2;\ D\}$
$\rightarrow \{S_1;\ E;\ (APP_1 \bullet S_1) \bullet randlist_2 \bullet APP_3 \bullet\ bv \bullet\ exp\text{-}seq \bullet C_2;\ D\}$

Comment: The lexical bindings are not lost and the operands are evaluated. Note the use of the composite meta-symbol $(APP_1 \bullet S_1)$ which serves to indicate that lexicals are to be used, and as a place-holder for the stack as it was before the argument evaluation was started. APP_3 is also used to avoid creating a *lambda-abstraction* from *bv* and *exp-seq* which would otherwise require the allocation of space. It is a design goal (not achieved by most LISP interpreters) that the evaluator should not wantonly consume heap-space. The question arises, what makes *lambda-expressions* used as operators deserve special treatment? The answer is that we wish to avoid the unnecessary closure formation that would otherwise occur.

8.1.1.2. Macro composition with explicit *mlambda-exp* as operator.
$\{\underline{\mathrm{MLAMBDA}} \bullet S;\ E;\ OP \bullet (\mathrm{y} \cdot (bv \cdot exp\text{-}seq)) \bullet OP \bullet (\mathrm{w} \cdot randlist_2) \bullet C_2;\ D\}$

8.1.1.2.1. If *bv* is conformal with $(\mathrm{w} \cdot randlist_2)$
$\rightarrow \{() \bullet ();\ \mathbf{bind}\{bv;\ (\mathrm{w} \cdot randlist_2);\ ((ehE \bullet lE) \bullet tE)\};\ ES \bullet exp\text{-}seq \bullet ();$
$\{S;\ E;\ EVAL \bullet C_2;\ D\}\}$

8.1.1.2.2. Otherwise $\rightarrow D_{macro\text{-}non\text{-}conformal}$.

Comment: The lexical bindings are available during the evaluation of the expression sequence *exp-seq*. Note that the original expression acts as the operand to the macro.

8.1.1.3. Explicit *mu-exp* as operator.
$\{\underline{\mathrm{MU}} \bullet S;\ E;\ OP \bullet (\mathrm{y} \cdot (bv \cdot randlist_3)) \bullet\ OP \bullet (\mathrm{w} \cdot randlist_2) \bullet C_2;\ D\}$
$\rightarrow \{S;\ E;\ (APP_1 \bullet S) \bullet randlist_3 \bullet APP_3 \bullet\ bv \bullet\ randlist_2 \bullet C_2;\ D\}$

Comment: A context closure is avoided here. Notice that:
$((\underline{\mathrm{LAMBDA}}\ bv_0 \cdot exp\text{-}seq_0) \cdot randlist_0) \equiv ((\underline{\mathrm{MU}}\ bv_0 \cdot randlist_0) \cdot exp\text{-}seq_0)$

8.1.1.4. Explicit <u>FR*CODE</u> operator.
$\{\underline{\mathrm{FR{*}CODE}} \bullet S;\ E;\ OP \bullet (\mathrm{y} \cdot (e_1\ z \ldots)) \bullet OP \bullet (\mathrm{w} \cdot randlist_2) \bullet C_2;\ D\}$
$\rightarrow \{S;\ E;\ e_1 \bullet OP \bullet (\mathrm{w} \cdot randlist_2) \bullet C_2;\ D\}$

Comment: *operator-code-exp* are of the form: $(e_1\ e_2\ f\text{-}list \cdot lap\text{-}code)$, where the value of e_1 is <u>FR*CODE</u>, and are equivalent to e_2 for the interpretive semantics. For compiled code it serves to define *e* by a sequence of *lap-stmts* where *lap-stmts* are LISP assembly program instructions, the definition of which is machine dependent and will not be discussed in this section. FR*CODE expressions provide an escape from the domain of LISP expressions for compiled code and can be used by systems programmers to provide efficient, compiled realizations of special expressions.

8.1.1.5. Explicit *seq-exp* as operator.
$\{\underline{\mathrm{SEQ}}\bullet S;\ E;\ OP\bullet(\mathrm{y}\cdot(\mathit{tag\ aux\ s}\ldots))\bullet OP\bullet(\mathrm{w}\cdot \mathit{randlist}_2)\bullet C_2;\ D\}$
$\rightarrow\{S;\ E;\ (APP_1\bullet S)\bullet \mathit{randlist}_2\bullet(SEQ_1\bullet S)\bullet(\mathit{tag\ aux\ s}\ldots)\bullet C_2;\ D\}$

Comment: Statement sequences provide a paradigm for algorithmic or procedural control flow concepts. This rule simply sets up to evaluate the values for the *aux* stack places, and then to apply the sequence. When applied the statements of the sequence are evaluated in a left to right manner (as are the operands of combinations) except that the control sequence may be changed through the use of the go-expressions. The labels, tags and auxiliaries have restricted lexical scopes, that is, only those sequence-expressions nested within a single binding contour have the usual conventions of lexical scope. To avoid confusion with the scope rules of lexical variables we shall refer to this as static scope. The operands are evaluated before the sequence is applied. Not calling the stack place names by the term variable may be a bit silly, but even though they may be used (in many cases) as analogs to variables defined by ***E***, they are not isomorphic. An important difference is that the places are not shared by other states.

8.1.2. Repeated operator evaluation, closure avoidance cases.
If $C_1 = (REAP\bullet S_1)\bullet\mathrm{u}\bullet C_2$ and $\mathrm{x} \in \{\underline{\mathrm{LAMBDA}} \mid \underline{\mathrm{FR{*}CODE}}\}$.

8.1.2.1. $\{\underline{\mathrm{LAMBDA}}\bullet S;\ E;$
$OP\bullet(\mathrm{y}\cdot(\mathit{bv}\cdot \mathit{exp\text{-}seq}))\bullet(REAP\bullet S_1)\bullet(\mathrm{w}\cdot \mathit{randlist}_2)\bullet C_2;\ D\}$
$\rightarrow\{S_1;\ E;\ (APP_2\bullet S_1)\bullet()\bullet APP_3\bullet \mathit{bv}\bullet \mathit{exp\text{-}seq}\bullet C_2;\ D\}$

8.1.2.2. $\{\underline{\mathrm{FR{*}CODE}}\bullet S;\ E;$
$OP\bullet(\mathrm{y}\cdot(e_1\ \mathrm{z}\ldots))\bullet(REAP\bullet S_1)\bullet(\mathrm{w}\cdot \mathit{randlist}_2)\bullet C_2;\ D\}$
$\rightarrow\{S;\ E;\ e_1\bullet C_1;\ D\}$

Comment: In the case of repeated evaluation of operators, closure formation can also be avoided if a *lambda-expression* pertains. Note the use of the composite meta-symbol $(APP_2\bullet S_1)$ which serves to indicate that lexicals are not to be used, and as a place-holder for the stack as it was before the argument evaluation was started. The lexical context is lost in the case of repeated evaluation. A *body* that arrives as part of a computed value is not considered lexically present. The FR*CODE could hide such a case, so it also must be treated specially.

8.1.3. Operator was not an explicit *abstraction-exp* but is an *sf*.
Otherwise, $\rightarrow\{\mathit{rand\text{-}list}_1\bullet S;\ E;\ SF\bullet\mathrm{x}\bullet C_1;\ D\}$

Comment: The usual practice in LISP systems is to avoid the operator evaluation entirely in these cases. That is, special forms as operators must be explicit, reserved identifiers. The compiler for these systems is free to have a fixed idea about their semantics. This system fixes its ideas about special form application expressions based on the value of the operator, likewise for macro's. We can always define or compile with respect to an environment that gives fixed values for the operators we wish to fix. The ultimate interpretation of special forms is delayed until application, special forms apply to the *randlist* of the *combination*.

8.2. Operator is a macro.

{x•*S; E;* ***OP***•(y • *randlist*$_l$)•***C***$_l$***; D***}
where x ={*mlambda-abstraction* | *mbpi* | *funarg-macro*}
where a *funarg-macro* is a *funarg* or *constant-closure* whose *e-part* is {*mlambda-abstraction* | *mbpi*}.
→{ (y •*randlist*$_l$) •***S; E; MAPP***•x•***EVAL***•***C***$_l$***; D***}

Comment: This rule elevates the status of macros. Macros apply to the *combination* form of which they were the operator. The result of the application (macro expansion) is then reevaluated.

8.3. Operator is a context abstraction.

{x•*S; E;* ***OP***•(y • *randlist*$_l$)•***C***$_l$***; D***}
where x ={*mu-abstraction* | *closed-context*}
where a *closed-context* is a *funarg* or *constant-closure* whose *e-part* is {*mu-abstraction* | *closed-context*}.
→{*rand-list*$_l$•***S; E; SF***•x•***C***$_l$***; D***}

Comment: This rule elevates the status of contexts. Contexts apply like *sf*.

8.4 Otherwise, ordinary application is presumed.

else→{***S; E;*** *(APP*$_2$•*S)*•*rand-list*$_l$•x•***C***$_l$***; D***}

Comment: The else-clause pertains in the case of ordinary application, in which case lexical bindings will tentatively be dropped during application.

Comment: The important thing being decided in rule 8 is whether ordinary application or macro application or special application is appropriate. In the cases of macro application the unevaluated original expression becomes the operand. The special forms are also recognized here and applied to their unevaluated *randlist*. If macro or special application is not indicated by the value of *rator* then ordinary application is presumed. At some cost in added complexity, the concept of lexicality and the wanton-heap-use avoidance principle have been introduced.

Macro Application

9. Macro application

{y•*S; E; MAPP•x•C; D*}

9.1. If x = *mbpi*

9.1.1. If **bv–of**{x} and y are conformal
→{*()*; **bind**{**bv–of**{x}; y; *((ehE)•E)*}; *lc*$_{entry}${x}; {*S; E; C; D*}}

9.1.2. Otherwise → *D*$_{non\text{-}conformal\text{-}app}$,

9.2. If x = %(<u>MLAMBDA</u> *bv* • *exp-seq*)

9.2.1. And y is conformal with *bv*
→{()•*()*; **bind**{*bv*; y; *((ehE)•E)*}; *ES•exp-seq•()*; {*S; E; C; D*}}

9.2.2. Otherwise → *D*$_{non\text{-}conformal\text{-}app}$,

9.3. If x is a *macro-funarg* with z the *e-part* and *sd* the *s-part* where *sd* has an *E*$_f$.

9.3.1. If z is *mbpi*

9.3.1.1. If **bv–of**{z} and y are conformal
→{*()*; **bind**{**bv–of**{z}; y; *((ehE•lE*$_f$*)•tE*$_f$*)*}; *lc*$_{entry}${z}; {*S; E; C; D*}}

9.3.1.2. Otherwise → *D*$_{macro\text{-}non\text{-}conformal}$,

9.3.2. If z = %(<u>MLAMBDA</u> *bv* • *exp-seq*)

9.3.2.1 If y is conformal with *bv*
→{()•*()*; **bind**{*bv*; y; *((ehE•lE*$_f$*)•tE*$_f$*)*}; *ES•exp-seq•()*;
{*S; E; C; D*}}

9.3.2.2. Otherwise → *D*$_{macro\text{-}non\text{-}conformal}$,

9.4. Otherwise → {y•*S; E; C; D*}

Comment: There are just two ways that this rule comes into action: One is after rule 8.2 the other is through the specific use of the basic function MDEFX. Note that in the case of applicable FUNARGs the lexical variables of the environment of the FUNARG are viable but in the case of *mbpi* and *mlambda-abstraction* lexical variables are dropped. The otherwise clause is interesting because it illustrates that if the macro being applied to the expression y is not a recognizably macro-applicable form then the expression y itself is returned unchanged. The otherwise clause only occurs through explicit use of MDEFX (see below).

Argument Evaluation (for Ordinary Application)

10. Evaluate Arguments Sequentially for Ordinary Application

$\{S;\ E;\ (\{APP_1\ /\ APP_2\}_3 \bullet S_1) \bullet (rand_1\ rand\ldots) \bullet C;\ D\}$
$\rightarrow \{S;\ E;\ rand_1 \bullet (\{APP_1\ /\ APP_2\}_3 \bullet S_1) \bullet (rand\ldots) \bullet C;\ D\}$

Comment: This rule entails the evaluation of operand expressions in left to right order. Note the use of the composite meta-symbol which serves both to indicate whether lexicals are to be used or not, and as a place-holder for the stack as it was before the argument evaluation was started.

Ordinary Application

11. Ordinary application

$\{a_n \bullet \ldots a_1 \bullet S;\ E;\ (\{APP_1\ /\ APP_2\}_3 \bullet S) \bullet c \bullet x \bullet C;\ D\}$

11.1 If x = APP_3 then

$\{a_n \bullet \ldots a_1 \bullet S;\ E;\ (\{APP_1\ /\ APP_2\}_3 \bullet S) \bullet c \bullet APP_3 \bullet\ bv \bullet\ exp\text{-}seq \bullet C_2;\ D\}$

11.1.1. If bv and $a_1 \ldots a_n$ are conformal
$\rightarrow \{() \bullet ();\ \mathrm{bind}_2\{bv;\ \{\ ((ehE \bullet lE) \bullet tE)\ |\ ((ehE) \bullet E)\ \}_3;\ a_1;\ldots a_n\};\ ES \bullet exp\text{-}seq \bullet ();$
$\{S;\ E;\ C_2;\ D\}\}$
11.1.2. Otherwise $\rightarrow D_{non\text{-}conformal\text{-}app}$.

Comment: APP_3 is used here as a means of using control stack space rather than heap space.

Lambda Abstractions

11.2 If x = %(<u>LAMBDA</u> $bv \cdot exp\text{-}seq$)

$\rightarrow \{a_n \bullet \ldots a_1 \bullet S;\ E;\ (\{APP_1\ /\ APP_2\}_3 \bullet S) \bullet c \bullet APP_3 \bullet\ bv \bullet\ exp\text{-}seq \bullet C;\ D\}$

Comment: See rule 11.1.

Function Binary Program Images

11.3 If x = *fbpi*

11.3.1. If **bv-of**{x} and $a_1 \ldots a_n$ are conformal
$\rightarrow$ {*()*; **bind**$_2${**bv-of**{x}; *((ehE)•E)*; $a_1;\ldots a_n$}; *lc*$_{entry}${x}; {*S; E; C; D*}}

11.3.2. Otherwise $\rightarrow$ *D*$_{non\text{-}conformal\text{-}app}$

Comment: Illustrates the application of a machine coded subroutine.

Distinguished basic functions

11.4 If x = *ur*, understood primitive operator constants. *UR* = {*FIX-UR* ∪ *MULT-UR*}.

Fixed number of argument understood operators

11.4.1. If x = *fix-ur* ∊ *UR*. It is a constant operator with definite number of arguments.

11.4.1.1. EVA1, the evaluate in the current environment operator.
{a_1•*S; E; ({APP*$_1$ / *APP*$_2$*}*$_1$*•S)*•()•<u>EVA1</u>•*C; D*}
$\rightarrow$ {*()*; *((ehE)•E)*; a_1•*()*; {*S; E; C; D*}}

Comment: Lexical bindings are not accessible to computed expressions. The only way to achieve this is to evaluate a constructed *funarg*. Constructed *funargs* are possible but through the use of a special *fix-ur*. Such an operator is necessary for LISP system programs, but is not a feature of the language being defined. It would violate the lexical scoping rules for the non-fluid variables. EVA1 achieves an optimization not possible by EVAL in that it does not require an attendant state saving.

11.4.1.2. If x is <u>MDEFX</u>, the operator for expanding macro's.
{a_2•a_1•*S; E; ({APP*$_1$ / *APP*$_2$*}*$_1$*•S)*•()•<u>MDEFX</u>•*C; D*}
$\rightarrow$ {a_2•*()*; *((ehE)•E)*; *MAPP*•a_1•*()*; {*S; E; C; D;* }}

Comment: This makes it possible for the LISP compiler (and indeed any function) to get one level of macro-expansion without the usually attendant evaluation.

11.4.1.3. If x is APPLX the operator for applying an operator to list computed (*arg*...)

$\{a_n \bullet \ldots a_1 \bullet S;\ E;\ (\{APP_1 \mid APP_2\}_3 \bullet S) \bullet () \bullet \underline{\mathrm{APPLX}} \bullet C;\ D\}$

where $a_2 = (a_{2.1} \ldots a_{2.n})$.

$\rightarrow \{a_1 \bullet a_{2.n} \bullet \ldots a_{2.1} \bullet S;\ E;\ (APP_2 \bullet S) \bullet () \bullet \underline{\mathrm{CALL}} \bullet C;\ D\}$

Comment: This allows the operand values to be computed as a list by an arbitrary LISP computation: A feature which could not in general be obtained otherwise.

11.4.1.4. If x is the EVAL, and a_1 and a_2 are present.

$\{a_n \bullet \ldots a_1 \bullet S;\ E;\ (\{APP_1 \mid APP_2\}_3 \bullet S) \bullet () \bullet \underline{\mathrm{EVAL}} \bullet C;\ D\}$

11.4.1.4.1. If a_2 is an *sd*

$\rightarrow \{();\ ((ehE) \bullet E_1);\ a_1 \bullet ();\ \{S;\ E;\ C;\ D\}\}$ where *sd* has E_1 as its ***E-part***.

11.4.1.4.2. Otherwise $\rightarrow \{\underline{\mathrm{EVAL}} \bullet a_2 \bullet a_1 \bullet S;\ E;$
$(APP_1 \bullet S) \bullet () \bullet \%(\underline{\mathrm{LAMBDA}}$?ARGS? ((ERR2 7) ?ARGS?))$\bullet C;\ D\}$.

Comment: This is the usual LISP evaluation with respect to a given environment. Note, the lexical dropping.

11.4.1.5. Otherwise, some understood basic operator.

11.4.1.5.1. If **bv-of{x}** and $a_1 \ldots a_n$ are conformal then

$\rightarrow \{x\{a_1;\ldots a_n;\ ((ehE) \bullet E)\} \bullet S;\ E;\ C;\ D\}$

11.4.1.5.2. Otherwise $\rightarrow D_{non\text{-}conformal\text{-}app}$,

Multiple argument understood operators

11.4.2. If x = *mult-ur*, it is a constant operator with an indefinite number of arguments.

11.4.2.1. CALL, the function to apply the operator which is computed after the operands to however many arguments were transmitted.

$\{a_n \bullet a_{n-1} \bullet \ldots a_1 \bullet S;\ E;\ (\{APP_1 \mid APP_2\}_3 \bullet S) \bullet () \bullet \underline{CALL} \bullet C;\ D\}$

11.4.2.1.1. If a_n exists and is a *fbpi*.

11.4.2.1.1.1. If **bv-of**$\{fbpi\}$ and $a_1 \ldots a_{n-1}$ are conformal.

$\rightarrow \{();\ \mathbf{bind}_2\{\mathbf{bv\text{-}of}\{fbpi\};\ ((ehE) \bullet E);\ a_1; \ldots a_{n-1}\};\ lc_{entry}\{fbpi\};$
$\{S;\ E;\ C;\ D\}\}$

11.4.2.1.1.2. Otherwise $\rightarrow D_{non\text{-}conformal\text{-}app'}$

11.4.2.1.2. If a_n exists and is %(FUNARG y *sd* • z) where *sd* has an E_l.

11.4.2.1.2.1. If y is *fbpi*.

11.4.2.1.2.1.1. If **bv-of**$\{fbpi\}$ and $a_1 \ldots a_{n-1}$ are conformal.

$\rightarrow \{();\ \mathbf{bind}_2\{\mathbf{bv\text{-}of}\{fbpi\};\ ((ehE \bullet lE_l) \bullet tE_l);\ a_1; \ldots a_{n-1}\};\ lc_{entry}\{fbpi\};$
$\{S;\ E;\ C;\ D\}\}$

11.4.2.1.2.1.2. Otherwise $\rightarrow D_{non\text{-}conformal\text{-}app'}$

11.4.2.1.2.2. Otherwise

$\rightarrow \{a_{n-1} \bullet \ldots a_1 \bullet ();\ ((ehE \bullet lE_l) \bullet tE_l);\ (APP_2 \bullet ()) \bullet () \bullet a_n \bullet ();$
$\{S;\ E;\ C;\ D\}\}$

11.4.2.1.3. Otherwise where a_n exists.

$\rightarrow \{a_{n-1} \bullet \ldots a_1 \bullet ();\ ((ehE \bullet lE_l) \bullet tE_l);\ (APP_2 \bullet ()) \bullet () \bullet a_n \bullet ();\ \{S;\ E;\ C;\ D\}\}$

11.4.2.1.4. Otherwise where a_n is not present.

$\{S;\ E;\ (\{APP_1 \mid APP_2\}_3 \bullet S) \bullet () \bullet \underline{CALL} \bullet C;\ D\}$

$\rightarrow \{() \bullet S;\ E;\ C;\ D\}$

Comment: This rule is ordinary calling in the current environment with arguments transmitted on the stack. Lexical bindings are dropped. CALL is used by the compiled code to attain certain efficiencies. In particular, the last argument a_n can be treated somewhat more efficiently than the rest of the arguments. Note the degenerate case: (CALL) = ().

11.4.2.2. If x is STATE the state saving operator it creates continuations

$\{a_n \bullet \ldots a_1 \bullet S;\ E;\ (\{APP_1 \mid APP_2\}_3 \bullet S) \bullet () \bullet \underline{STATE} \bullet C;\ D\}$

$\rightarrow \{sd \bullet S;\ E;\ C;\ D\}$

where *sd* has $\{S;\ E;\ C;\ D\}$

Comment: This *sd* constructing operator can only capture the current *S, E, C,* and *D*. The optional argument a_1 if present is *glonot* (see section on global environments). Likewise, a_2 if present is *glolst* (see section on global environments). These optional arguments allow the saved state to pertain to global environments other than the one current. If neither is specified then the current global context is assumed.

11.4.2.3. Otherwise, some understood basic operator.

11.4.2.3.1. If **bv-of**{x} and $a_1 \ldots a_n$ are conformal then

$\rightarrow \{x\{a_1; \ldots a_n;\ ((ehE) \bullet E)\} \bullet S;\ E;\ C;\ D\}$

11.4.2.3.2. Otherwise $\rightarrow D_{non\text{-}conformal\text{-}app'}$

Dynamic Macros

11.5. If x = { *mbpi* | *mlambda-abstraction*} → $D_{macro\text{-}inapplicable}$.

Comment: This would seemingly limit the free choice between macro definition and function definition styles for operators but it can be over come by clever use of the error channel (see section on Program Events). In any case we detect that it is too late to macro apply.

Closure Application

11.6. If x = *funarg* or *constant-closure* and y is the *e-part* and *sd* the *s-part*, where *sd* has E_l as its ***E-part*** then

11.6.1. If y is an *fbpi*.

11.6.1.1. If **bv–of**{y} and $a_1 \ldots a_n$ are conformal then
→ {*()*; **bind**$_2${**bv–of**{y}; *((ehE•lE$_l$)•tE$_l$)*; a_1; ...a_n}; *lc*$_{entry}${y}; {*S; E; C; D*}}

11.6.1.2. Otherwise → $D_{non\text{-}conformal\text{-}app}$'

11.6.2. Otherwise → {a_n•...a_1•*()*; E_l; *(APP$_l$•())*•()•y•*()*; {*S; E; C; D*}}

Comment: Notice the possible inconstancy of the *constant-closure*.

State Application

11.7 If x = *sd* where *sd* has {S_2; E_2; C_2; D_2} → {a_1•S_2; E_2; C_2; D_2}

Comment: A fact that this meta-language may not adequately convey is that S_2 and C_2 of the *sd* are copies but E_2 and D_2 are shared references. This illustrates the "continuation" of a state.

Code Abstraction Application

11.8. *operator-code-abstraction*.

{a_n•...a_1•*S; E; ({APP$_1$ / APP$_2$}$_3$•S)*•()•%(<u>FR*CODE</u> e_l z...)•*C; D*}
→{a_n•...a_1•*S; E; ({APP$_1$ / APP$_2$}$_3$•S)* •()•e_l•*C; D*}

Comment: This expression form is meant for the exploitation of the compiler and is included here only to reveal its import to the interpreter. *operator-code-abstractions* provide an escape, for compiled code, from the domain of LISP expressions, and into the domain of LAP code. They can be used by systems programmers to provide efficient, compiled realizations of basic operators or to provide access to system function not normally provided for by LISP semantics. They are normally used in operator position. It is hoped that such devices will be used only by the well informed and well intended.

Sequence application

11.9. If x = $(\{SEQ_1 \,/\, SEQ_2\} \bullet S)$
$\{a_n\bullet \ldots a_1\bullet S;\ E;\ (\{APP_1 \,/\, APP_2\}_3\bullet S)\bullet c\bullet (\{SEQ_1 \,/\, SEQ_2\} \bullet S)\bullet\ (tag\ aux\ s\ldots) \bullet C_2;\ D\}$

11.9.1. If *aux* and $a_1 \ldots a_n$ are conformal
$\rightarrow \{()\bullet a_n\bullet \ldots a_1\bullet S;\ E;\ \{(\{SEQ_1 \,/\, SEQ_2\} \bullet S)\bullet\ (tag\ aux\ s\ldots) \bullet (s\ldots) \bullet C_2;\ D\}$
11.9.2. Otherwise $\rightarrow D_{non\text{-}conformal\text{-}app}$.

Comment: Central rule for the conformal mapping of *aux* on to the parameters.

Sequence Abstractions

11.10 If x = %($\underline{SEQ}$ • (*tag aux s...*))

$\rightarrow \{a_n\bullet \ldots a_1\bullet S;\ E;\ (\{APP_1 \,/\, APP_2\}_3\bullet S)\bullet c\bullet (SEQ_2\bullet S)\bullet\ (tag\ aux\ s\ldots) \bullet C_2;\ D\}$

Comment: Notice that SEQ_2 is categorically used here, this will serve as a static scope stopper. A quoted or any constant or any computed *seq-abstraction* does not have any inherited static context.

Inapplicable Objects

11.11. If x $\in$ $\{SF \mid MU\text{-}ABSTRACTION \mid CLOSED\text{-}CONTEXT\}$ $\rightarrow D_{sf\text{-}inapplicable}$.

Comment: Here again we chose to intercept a case where it is now too late to not evaluate the operands. In other words, having decided to do ordinary application we find that the operator requires special forms application and that is inconsistent, therefore to be treated as an error.

11.12. Place-holder assignment (see rule 20.)

11.13. Assignment (see rule 19.)

11.14. If x is a constant not explicitly mentioned elsewhere in rule 11 then

$\rightarrow$ $D_{inapplicable\text{-}object}$

Comment: The error channel for $D_{inapplicable\text{-}object}$ can be replaced (by a function to produce (x a_1 ... a_n) for instance.) Traps are discussed is greater detail in a following section. This is an instance where the model exposes the computational import of a semantics question. The error trap channels provide flexibility but require wanton CONSing and the creation of additional activation records.

Operator evaluation repeated

11.15. Operator not directly applicable.

Otherwise, $\rightarrow$ {a_n•...a_1•*()*; *((ehE)•E)*; x•*(REAP•())*•x•*()*; {*S; E; C; D*}}

Comment: Application of x to the operand values is sometimes referred to as ordinary application. When x is a macro or meta form, an error break occurs. When x is a FUNARG, a new state must be created. When x is a constant form unknown to Rule 11, an error break occurs. The otherwise clause is interesting because it illustrates the repeated evaluation of the operator form until the applicable function it denotes is revealed.

Reapply reevaluated operator

12. Reapply reevaluated operator.

{z•a_n...a_1•*S; E; (REAP•S)*•x•*C; D*}

12.1. If x=z$\rightarrow$ $D_{inapplicable\text{-}object}$

12.2. Otherwise $\rightarrow${a_n•...a_1•*S; E; (APP•S)*•()•z•*C; D*}

Comment: Reevaluation results in the dropping of lexical variables. Readers may reasonably find fault with the trivial loop detection case semantics.

Meta Applicable Forms

13. Special Forms application.

$\{a_1 \bullet S;\ E;\ SF \bullet x \bullet C;\ D\}$

13.1. Conditional expressions.

$\{a_1 \bullet S;\ E;\ SF \bullet \underline{COND} \bullet C;\ D\} \rightarrow \{() \bullet S;\ E;\ \boldsymbol{PRED} \bullet () \bullet a_1 \bullet C;\ D\}$

Comment: ***PRED*** is merely the meta-linguistic cog for the conditional. The balance of the rules for the conditional are given later.

13.2. Auxiliary stack-place values.

$\{a_1 \bullet S;\ E;\ SF \bullet \underline{AUX} \bullet C;\ D\}$ where $a_1 = (x\ \ldots)$.

13.2.1. If $C = \ldots y \ldots \bullet (\{SEQ_1 \mid SEQ_2\} \bullet S_1) \bullet (tag\ aux\ s \ldots) \bullet C_2$
where $y \neq (SEQ_2 \bullet S_3)$
and $aux = (aux\text{-}id \ldots)$ and $x = aux\text{-}id_j$
and $S_1 = \ldots \bullet a_i \ldots \bullet a_j \ldots \bullet a_0 \bullet S_4$ then
$\rightarrow \{a_j \bullet S;\ E;\ C;\ D\}$

13.2.2. Otherwise $\rightarrow D_{unbound\text{-}AUX}$

Comment: The basic access operator for the contents of a stack-place. Notice that there is no access scope beyond the innermost computed sequence (indicated by $(SEQ_2 \bullet S_3)$). The reader may well ask: why was it necessary to introduce yet another class of variables? The reply to that question is that S has some attractive properties that we wish to exploit. Namely:
1. With respect to the rules of this semantics we notice that the constituents of S are staticly determined from the point of view of C. This means that compiled references may be early bound to constant offsets.
2. In the state saving and continuation of a state the values on S are not shared as in the case of E. Because they are copied rather than referenced they are "undone".

13.3. Enter expression sequence.

$\{a_1 \bullet S;\ E;\ SF \bullet \underline{PROGN} \bullet C;\ D\} \rightarrow \{() \bullet S;\ E;\ ES \bullet a_1 \bullet C;\ D\}$

Comment: This form is an applicable expression sequence operator. The balance of the rules for expression sequence evaluation are given later (see rule 17).

13.4. Return expression.

$\{a_1 \bullet S;\ E;\ SF \bullet \underline{RETURN} \bullet C;\ D\} \rightarrow \{S;\ E;\ \{e \mid ()\}_1 \bullet ();\ D\}$
where $a_1 = \{(e \cdot z) \mid atom\}_1$.

Comment: Allowed anywhere a variable is allowed. Causes return with value from the current state frame.

13.5. EXIT expression.

$\{a_1 \bullet S;\ E;\ SF \bullet \underline{\mathrm{EXIT}} \bullet C;\ D\} \rightarrow \{S;\ E;\ e \bullet EXIT \bullet tag \bullet C;\ D\}$
where $a_1 = (e \cdot tag)$,

Comment: The action of this rule is to cause *e* to be evaluated like any other operand expression of the current static statement context. Should that evaluation produce a value then rule 18 will complete the exit. An identifier argument to EXIT is treated as a variable not as a statement label.

13.6. Go expression.

$\{a_1 \bullet S;\ E;\ SF \bullet \underline{\mathrm{GO}} \bullet C;\ D\}$
where $a_1 = (st\text{-}lab_l \cdot \mathrm{v})$.
13.6.1. If $C = \ldots \mathrm{y} \ldots \bullet (\{SEQ_l \mid SEQ_2\} \bullet S_l) \bullet (tag\ aux\ s\ldots)\ (s_l \ldots) \bullet C_l$.
where $\mathrm{y} \neq (SEQ_2 \bullet S_3)$
where $(s_l \ldots) = (\ldots st\text{-}lab_l\ s_2 \ldots)$.
and S must be $\ldots \bullet S_l$
$\rightarrow \{S_l;\ E;\ (\{SEQ_l \mid SEQ_2\} \bullet S_l) \bullet (st\text{-}lab_l\ s_2 \ldots) \bullet (s_l \ldots) \bullet C_l;\ D\}$
13.6.2. Otherwise,
$\rightarrow \{st\text{-}lab_l \bullet S;\ E;\ (APP_l \bullet S_l) \bullet () \bullet \%(\underline{\mathrm{LAMBDA}}(?\mathrm{ARGS}?)$
$(\mathrm{UNWIND}((\mathrm{ERR2}\ 10)\ ?\mathrm{ARGS}?))) \bullet C;\ D\}$

Comment: GO expressions affect only the control and stack. Which is, the principal reason for a distinguished GO.

13.7. Closure forming expression.

$\{a_1 \bullet S;\ E;\ SF \bullet \mathrm{x} \bullet C;\ D\}$
where x is $\{\underline{\mathrm{FUNCTION}} \mid \underline{\mathrm{LAMBDA}} \mid \underline{\mathrm{MLAMBDA}} \mid \underline{\mathrm{FR{*}CODE}} \mid \underline{\mathrm{SEQ}}\}_2$
and $a_1 = (\mathrm{y} \cdot \mathrm{w})$ then
13.7.1. If x is FUNCTION or FR*CODE and y is $funarg_l$
then $\rightarrow \{funarg_l \bullet S;\ E;\ C;\ D\}$
13.7.2. Otherwise, $\rightarrow \{funarg_2 \bullet S;\ E;\ C;\ D\}$.
where $funarg_2$ is created and has u as *e-part* and *sd* as *s-part*
where u is:
$\{ \mathrm{y} \mid$
$\%(\underline{\mathrm{LAMBDA}}\ \mathrm{y} \cdot \mathrm{w}) \mid$
$\%(\underline{\mathrm{MLAMBDA}}\ \mathrm{y} \cdot \mathrm{w}) \mid$
$\%(\underline{\mathrm{FR{*}CODE}}\ \mathrm{y} \cdot \mathrm{w}) \mid$
$\%(\underline{\mathrm{SEQ}}\ \mathrm{y} \cdot \mathrm{w}) \}_2$.
and *sd* has ***E*** as its ***E-part***.
i.e., the lexical bindings (if present) are operative in the environment of the closure.

Comment: The *lambda-expressions* and *mlambda-expressions* of LISP are prototypical functions and macros; upon evaluation their meaning is bound and they become functions and macros. The *funarg* (closure) contains the abstraction and the

Prof. Dr. H. Stoyan
Universität Erlangen-Nürnberg
Institut für Mathematische Maschinen
und Datenverarbeitung (Informatik VIII)
Am Weichselgarten 9
91058 Erlangen

environment that binds the variables. Note that the environments can be shared, indeed any value object can be shared. Assignment and updating can therefore change the meaning of a variable. From this we conclude that meaning is not closed (in the mathematical sense), only bound, by these closures. We will nevertheless refer to them as closures (in a computational sense).

For the present model the representation of *sd* will not be defined. Only the *E* component is needed by the FUNARG device. These *funarg's* and *sd's* require allocation, a sufficient reason to avoid the overuse of this mechanism. The <u>FUNCTION</u> device is most useful when we wish to pass a functional object around by name, i.e. *e* is an *id*. It may be used to close any expression *e* with respect to *E*. The FUNARG device is used to represent these closures.

Note that extraneous operands are ignored. Observe that (<u>FUNCTION</u> *mlambda-exp*) is unproductive (will produce only "dynamic macro not allowed" condition), but (<u>FUNCTION</u> *lambda-exp*) is productive but unnecessary.

13.8. <u>LABEL</u> expressions, evaluate with respect to a dummy environment.

$\{a_1 \bullet S;\ E;\ SF \bullet \underline{\text{LABEL}} \bullet C;\ D\} \rightarrow \{bv \bullet body \bullet ();\ E_l;\ body \bullet LABEL \bullet ();\ \{S;\ E;\ C;\ D\}\}$
where a_1 is (*bv body* • z).
where E_l = **bind**{*bv*; **du**{*bv*}; ((*ehE*•*lE*)•*tE*)}

du{*bv*} = *bv* if *bv* is a *c*.
(Θ•Θ) if *bv* is an *ident*
(**du**{bv_l} • **du**{bv_2}) if *bv* is (bv_l • bv_2)
all other cases are undefined.

Comment: Label-expressions allow the computation of objects with shared references without requiring the user to use the "dangerous" update operators. It may be used to create recursive functions, mutually recursive functions, and list structures with shared references. The second part of <u>LABEL</u> expression evaluation, the fixup phase, is found in rule 21. The problem that this operator only partially solves is to find solutions to all equations of the form: *bv* = *e*. This technique works only if a "guessed" structure of the same shape as *bv* with the variables denoting pairs†, is conformal to the value of *e* computed in a context where *bv* is bound to that structure. In other words the solution should conform to *bv* and to *e* and the each variable of *bv* should be bound to its conformal component of the solution, and the variable references in *e* should compute the value denoted by that variable in the conformal mapping of *bv* on the solution.

† In the case of restricted type variables a better guess can be made.

13.9. SETQ.

$\{a_1 \bullet S;\ E;\ SF \bullet \underline{\mathrm{SETQ}} \bullet C;\ D\} \rightarrow \{() \bullet id \bullet S;\ E;\ ES \bullet (e\ldots) \bullet (APP_1 \bullet S) \bullet () \bullet \underline{\mathrm{SET}} \bullet C;\ D\}$
where a_1 is $(id\ e)$.

Comment: SETQ will assign to lexical variables because of the use of APP_1. This is a consequence of the *ur* SET which is described in rule 19, and which this rule uses to accomplish its purpose.

13.10. Quotation.

$\{a_1 \bullet S;\ E;\ SF \bullet \underline{\mathrm{QUOTE}} \bullet C;\ D\} \rightarrow \{s\text{-}exp_1 \bullet S;\ E;\ C;\ D\}$
where a_1 is $(s\text{-}exp_1 \bullet s\text{-}exp_2)$.

Comment: In order to denote the non-idempotent forms (pairs, funargs and identifiers) the quotation device is necessary in LISP. This follows from the insistence that expressions be data. Backus[2] points out "where meaning is not idempotent, we have chains of meanings, e.g.: (QUOTE *e*) → *e* → f ... etc.". In LISP such repeated evaluation is implicit only for operators during ordinary application, normally expression evaluation is one level of reduction. Notice that operator expression quotation has the meaningful effect of assuring ordinary application without lexical access!

13.11. Context application

$\{a_1 \bullet S;\ E;\ SF \bullet \%(\underline{\mathrm{MU}}\ bv \bullet valuelist) \bullet C;\ D\}$
$\rightarrow \{v_n \bullet \ldots v_1 \bullet S;\ E;\ (APP_2 \bullet S) \bullet () \bullet APP_3 \bullet bv \bullet a_1 \bullet C;\ D\}$
where $valuelist = (v_1 \ldots v_n)$.

Comment: Application of a context abstraction is explained in terms of ordinary application. Observe that such an applicable constant allows the imposition of a limited set of bindings into the current context with out requiring state saving. Note: The lexical variables (outside of *bv*) are inaccessible when a_1 finally evaluates. (%(MU *bv* • *valuelist*) • *exp-seq*) is equivalent to:
(APPLX %(LAMBDA *bv* • *exp-seq*) (QUOTE *valuelist*)).

13.12. Meta application of a *context-closure*

$\{a_1 \bullet S;\ E;\ SF \bullet \%(\underline{\mathrm{FUNARG}}\ \%(\underline{\mathrm{MU}}\ bv \bullet valuelist) \bullet sd_1) \bullet C;\ D\}$
$\rightarrow \{v_n \bullet \ldots v_1 \bullet S;\ E_1;\ (APP_1 \bullet ()) \bullet () \bullet APP_3 \bullet bv \bullet a_1 \bullet ();\ \{S;\ E;\ C;\ D\}\}$
where $valuelist = (v_1 \ldots v_n)$ and sd_1 has E_1 as its *E-part*.

Comment: This is the case of a context closure application where the surrounding lexical context is accessible. More powerful than applying EVAL to (PROGN • a_1) and a state denoting E_1. It is debatable whether $(APP_2 \bullet ())$ rather than $(APP_1 \bullet ())$ is appropriate; in which case lexical access would be limited to within the scope of the *bv* of the context-abstraction.

13.13. Context closure forming

$\{a_1 \bullet S;\ E;\ SF \bullet \underline{\text{MU}} \bullet C;\ D\}$
$\rightarrow \{bv \bullet S;\ E;\ (\underline{\text{STATE}}) \bullet (MU \bullet S) \bullet\ randlist \bullet C;\ D\}$
where $a_1 = (bv \bullet randlist)$

Comment: Ultimately evaluates to %(<u>FUNARG</u> %(<u>MU</u> *bv* • *valuelist*) • *sd*) see rule 14. From the point of view of a simple understanding of "lexical variables", this operator is the forbidden fruit. The subsequent application of the *funarg* could behave as if the computed expression occurred lexically in the same context as the *mu-expression* that formed the *closed-context*. The safety of lexical variables is therefore threatened by the <u>MU</u> operator. Were it not for this operator there would be no possibility to update the lexicals with operations invisible to the lexical text. The ability to treat lexical variables as place holders, subject to renaming (alpha-conversion), removal from *E*, and binding contour flattening, is thus complicated by the existence of this operator. These optimizations may not be practiced if a *mu-expression* occurs in the context.

13.14. <u>SETX</u>.

$\{a_1 \bullet S;\ E;\ SF \bullet \underline{\text{SETX}} \bullet C;\ D\}$
$\rightarrow \{() \bullet id \bullet S;\ E;\ ES \bullet (e\ldots) \bullet (APP_l \bullet S) \bullet () \bullet AUXSET \bullet C;\ D\}$
where a_1 is $(id\ e)$.

Comment: <u>SETX</u> will assign to stack-place variables. See rule 20.

13.15. Ill-formed *sf*.

$\{a_1 \bullet S;\ E;\ SF \bullet x \bullet C;\ D\} \rightarrow D_{\text{ill-formed}}$.
where x is *sf* but none of the above applies

Comment: Rule 13 could be viewed as a subroutine of rule 8.

Closed context preparation

14. $\{v_n \bullet \ldots v_1 \bullet sd_l \bullet bv \bullet S;\ E;\ (MU \bullet S) \bullet x \bullet C;\ D\}$

14.1 If x = () then,

$\rightarrow \{\%(\underline{\text{FUNARG}}\ \%(\underline{\text{MU}}\ bv\ v_1 \ldots v_n) \bullet sd_l) \bullet S;\ E;\ C;\ D\}$

Comment: A closed context is formed, it will apply specially to an expression that wil have access to the lexical of sd_l if there are any.

14.2 If x ≠ () then prepare the value of the next *rand*.

$\{v_n \bullet \ldots v_1 sd_l \bullet bv \bullet S;\ E;\ (MU \bullet S) \bullet (rand_l\ rand \ldots) \bullet C;\ D\}$
$\rightarrow \{v_n \bullet \ldots v_1 sd_l \bullet bv \bullet S;\ E;\ rand_l \bullet (MU \bullet S) \bullet (rand \ldots) \bullet C;\ D\}$

Comment: The values for the context are computed in left to right order, in the context that the context-expression occurred at the time the context closure was formed.

Conditional Expression

15. If ***PRED*** is at the head of control, it indicates completion of predicate,

$\{u \bullet S;\ E;\ PRED \bullet (e_l \ldots) \bullet x \bullet C;\ D\}$

15.1. Continue on to next predicate consequent, if u is (),

15.1.1. If x is $((p_2\ e_2 \ldots)(p_3\ e_3 \ldots) \ldots)$
$\rightarrow \{S;\ E;\ p_2 \bullet PRED \bullet (e_2 \ldots) \bullet ((p_3\ e_3 \ldots) \ldots) \bullet C;\ D\}$

15.1.2. If x is (*atom* • y)
$\rightarrow \{() \bullet S;\ E;\ PRED \bullet (e_l \ldots) \bullet y \bullet C;\ D\}$

15.1.3. If x is *atom*
$\rightarrow \{() \bullet S;\ E;\ C;\ D\}$

15.2. If u ≠ (), evaluate the consequent

15.2.1. If $\{u \bullet S;\ E;\ PRED \bullet (e_l\ e_2 \ldots) \bullet y \bullet C;\ D\}$ where u ≠ ().
$\rightarrow \{u \bullet S;\ E;\ ES \bullet (e_l\ e_2 \ldots) \bullet C;\ D\}$

Comment: The possibility of multiple consequents is known in some LISP system as the implied-PROGN feature. It can be considered as syntactic sugaring.

15.2.2. If $\{u \bullet S;\ E;\ PRED \bullet atom \bullet ((p_2\ e_2 \ldots) \ldots) \bullet C;\ D\}$ where $u \neq ()$.
$\rightarrow \{u \bullet S;\ E;\ C;\ D\}$

Comment: This is the case where no consequent is present, the predicate value is then the value of the conditional.

Statement Sequence Evaluation

16. Statement sequence evaluation, if $(SEQ_{\{1|2\}} \bullet S)$ at head of control.

$\{x \bullet S;\ E;\ (SEQ_{\{1|2\}} \bullet S) \bullet (tag\ aux\ s_1 \ldots) \bullet u \bullet C;\ D\}$

Comment: This form and its precursor PROG (now seen to be non-quintessential) has been much maligned as not part of "pure LISP". In fact it is of great teleological value because the common control structures are derivable from it and its attendant GO and EXIT and statement label forms. $(SEQ_{\{1|2\}} \bullet S)$ is merely the meta-linguistic cog for statement context evaluation semantics.

16.1. If u is *atom*, then leave sequence with value of last program statement.

$\rightarrow \{x \bullet S;\ E;\ C;\ D\}$

16.2. Statement labels are skipped, if u = (*st-lab s*...), then
$\{y \bullet S;\ E;\ (SEQ_{\{1|2\}} \bullet S) \bullet z \bullet (st\text{-}lab\ s \ldots) \bullet C;\ D\}$
$\rightarrow \{() \bullet S;\ E;\ (SEQ_{\{1|2\}} \bullet S) \bullet z \bullet (s \ldots) \bullet C;\ D\}$

Comment: No evaluation occurs for identifiers which occur as statement labels. Identifiers which occur as consequents of conditionals and identifiers which occur as arguments to exits are variables not statement labels. The fact that statement labels have the value NIL rather than the previous retained value is deliberate. The reason for this is that we wish be able to optimize the compilation of SEQ. In particular, we presently have no means of preserving the last computed value after a branch in a conditional to a final statement label.

16.3. If u = (*ps* • (*s*...)), program statements are evaluated sequentially.
$\{x \bullet S;\ E;\ (SEQ_{\{1|2\}} \bullet S) \bullet z \bullet (\ ps \bullet (s \ldots))\ \bullet C;\ D\}$
$\rightarrow \{S;\ E;\ ps \bullet (SEQ_{\{1|2\}} \bullet S) \bullet z \bullet (s \ldots)\ \bullet C;\ D\}$

Comment: The statements are evaluated in sequence, the value of the previous statement is not available during the current statements evaluation.

Expression Sequence Evaluation

17. If *ES* at head of control, then expression sequence evaluation.

$\{x \bullet S;\ E;\ ES \bullet u \bullet C;\ D\}$

17.1. Termination condition, if u is *atom*,

$\{x \bullet S;\ E;\ ES \bullet atom \bullet C;\ D\} \rightarrow \{x \bullet S;\ E;\ C;\ D\}$

Comment: The retained last expression value is the value of the sequence.

17.2. If u = $(e_1\, e_2 \ldots)$, evaluate next expression in an expression sequence.

$\{x \bullet S;\ E;\ ES \bullet (e_1\, e_2 \ldots) \bullet C;\ D\} \rightarrow \{S;\ E;\ e_1 \bullet ES \bullet (e_2 \ldots) \bullet C;\ D\}$

Comment: Each expression of the expression sequence is evaluated in sequence without access to the previous expression value.

Exit sequence expression

18. $\{x \bullet S;\ E;\ EXIT \bullet tag_1 \bullet C;\ D\}$

    18.1. If $C = \ldots y \ldots (SEQ_{[1|2]} \bullet S) \bullet (tag_1\ aux\ s_1 \ldots) \bullet u \bullet C_1;\ D\}$
    and S must be $\ldots \bullet S_1$,
    and $y \neq (SEQ_2 \bullet S_2)$
    then $\rightarrow \{x \bullet S_1;\ E;\ C_1;\ D\}$

    18.2. Otherwise, $\rightarrow D_{exit\text{-}error}$.

Comment: The action of rule 13.5 has evaluated the argument with respect to the static scope of the current sequence, in the case currently under consideration that evaluation completed without changing the control state, and the computed value is now to be returned as the value of the enclosing sequence with the matching *tag*.

Assignment

19. Assignment (This rule could have been 11.13 but was left out so that the memory *M* need not be introduced until it was required.)

{x•y•*S; E; (APP*$_{1|2}$*•S)*•()•$\underline{\text{SET}}$•*C; D*}*M*$_1$

19.1. If y is an *id*

→{{*E*{y;ASSIGN;x}$_1$ | **fluid**{y;ASSIGN;x}$_2$}•*S; E; C; D*}*M*$_2$
and {*E*{y;LOOKUP;()}$_1$ | **fluid**{y;LOOKUP;()}$_2$} *M*$_2$ = x

Comment: Only in the case that *APP*$_1$ pertains are lexical bindings sought.

19.2. If y is a *funarg* which has z as the *e-part* and *E*$_1$ as the ***E-part*** of the *s-part* then
→{x•z•*()*; *E*$_1$; *(APP*$_1$*•())*•()•SET•*()*; {*S; E; C; D*}}

19.3. Otherwise → {$\underline{\text{SET}}$•x•y•*S; E; (APP*$_1$*•S)*•()• %($\underline{\text{LAMBDA}}$?ARGS? ((ERR2 11)?ARGS?))•*C; D*} .

20. Place-holder assignment (see rule 11.12. and 13.14.)

{z•*id*$_1$•*S; E; (APP*$_1$*•S)* •()•*AUXSET*•*C; D*}*M*$_1$
20.1. If *C* = ...y...• *({SEQ*$_1$ / *SEQ*$_2$*}* •*S*$_1$*)*• (*tag aux*$_1$ *s*...) • *C*$_2$
where y ≠ *(SEQ*$_2$*•S*$_3$*)*
and *aux*$_1$ = (*aux-id*...) and *id*$_1$ = *aux-id*$_j$
and *S*$_1${*M*$_1$} = ...•a$_i$...•a$_j$...•a$_0$ •*S*$_4$ then
→ {z•*S; E; C; D*}*M*$_2$ and *S*$_1${*M*$_2$} = ...•a$_i$...•z•a$_{j-1}$...•a$_0$ •*S*$_4$

20.2. Otherwise → *D*$_{unbound\text{-}AUXSET}$

Comment: The basic update operator for the contents of a stack-place.

LABEL Operator

21. LABEL environment fixup phase.

```
{x•bv•body•S; $E_l$; LABEL•C; D} $M_l$
→{refix{bv; x; fixup{$E_l$; bv; x}$M_l$}•S; C; D} $M_n$

fixup{$E_l$; bv; x} $M_i$ = $E_l${M}$_n$
fixup : $E \times BV \times S\text{-}EXP \times M \rightarrow S\text{-}EXP \times E \times M$.

Where fixup{$E_l$; bv; x} $M_i$ =
    if bv is a c then $E \times M_i$,
    if bv is an ident i.e. {(FLUID $id_l$) | (LEX $id_l$) | $id_l$} then
        if { x $\notin$ $VALUES_{created\ by\ body}$ | x $\in$ ATOM } then $setq_{EandM}${$id_l$; x}
        if x is a pair x=($x_1$ • $x_2$) then
          $rplacd_{EandM}${ rplaca{$E_l${$id_l$; LOOKUP; ()}; $x_1$}; $x_2$}
    if bv is ($bv_l$ • $bv_2$) then
        if x $\in$ ATOM then undefined for now
        if x = ($x_l$ • $x_2$) then fixup{fixup{$E_l$; $bv_l$; $x_l$}; $bv_2$; $x_2$}
    all other cases are undefined.
```

The meta operators **setq**, **rplacd**, **rplaca**, and *E* are the rather obvious functions whose value domain is *{ S-EXP × E × M}*. Normally it will be sufficient to ignore the *E* and *M* aspects of such functions and they will be simply thought of as denoting *s-exp*. Occasionally (as is the case above) the *E* and *M* are the domains of interest and are indicated as above by the subscripting. It is generally tiresome to continually include the environment and memory in the domain and range considerations of all functions and so we are prone to leave them out. The reader is expected to assume that they are intended and to tell from context whether they are relevant.

```
refix{bv; x; $E_l$} =
    if bv is a c then c
    if bv is an ident i.e. {(FLUID $id_l$) | (LEX $id_l$) | $id_l$} then $E_l${$id_l$; LOOKUP; ()}
    if bv is a pair ($bv_l$ • $bv_2$) and x is a pair ($x_1$ • $x_2$) then
        rplacd{rplaca{x; refix{$bv_l$; $x_1$; $E_l$}}; refix{$bv_2$; $x_2$; $E_l$}}
    else undefined.
```

Comment: Having evaluated the *body* of the LABEL expression with respect to an environment in which the elements of *bv* were bound to dummy pairs, we now update those pairs under the assumption that the value x is an object of the same shape as *bv*. It has been suggested that if the initial guess for the dummy bindings leads to an undefined case during **fixup**, the actual value delivered should be chosen as a new dummy and the evaluate phase repeated, etc. The complete LABEL for typed variables will generate dummy values of the indicated kind. A word about the purpose of all this: We wish to compute self referring structures and LABEL provides that ability. So does SETQ, SET, RPLACA and RPLACD. Our definition of LABEL uses the meta linguistic equivalents of these operators. Why not just stick to the update operators? The answer is that the update operators can alter previously computed values thus changing their interpretation. Label does not have that property. It is not as dangerous!

THE INTERRUPT SYSTEM

Interrupts are external events rather that objects of the system, furthermore their detection is usually provided for in the underlying computing engine. This section posits a model that is tentative, and while it meets certain current practical demands, may serve as a start point for the development of a better model.

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. It dispatches the interrupt service function 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. It is basically 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.

In the case of the ***SECD*** machine we must extend our description to encompass these events. This can be done in the following manner:

A new meta-linguistic state component is introduced which is nothing more than some special storage which we shall view as a kind of circular list:

I=((*priority-level*$_1$ n_1 • k_1)... (*priority-level*$_s$ n_s • k_s)• I)

where *priority-level*$_i$ is a small positive integer that determines the priority of the interrupts of type k_i.
and n_i is a count of how may unserviced interrupts of the type k_i are pending,
and k_i is an integer which identifies a type or class of interrupt.

For example, I currently accommodates 7 unique interrupt classes.

I=((1 0 • 1) (3 0 • 2) (1 0 • 3) (5 0 • 4) (5 0 • 5) (3 0 • 6) (1 0 • 7)•I).

The purpose or definition of each interrupt class is given by Table 1.

This would suffice if interrupts had only to announce their occurrence. It is however the case that when certain interrupts occur they bring some data with them. This data must be enqueued in some manner. An association of such queues to interrupt types is provided by the following addition to the LISP machine:

R is a vector whose k^{th} element is a reference to a LISP data object in the LISP heap which defines the queue of pending data for pending interrupts of type *k*.

The data object representing such queues is a vector of non-pointer fixed-point numbers. ***R*** has associated with each element two index numbers which serve to define the place that new data is fed into the vector by interrupts and the place that data is next to be eaten by service functions. These indices wrap around as they are advanced beyond the capacity of the vector. In the case where the feed index would advance to the eat index the interrupt is lost and the index advance does not occur. The function NEW-QUEUE is provide to allow the user to redefine any particular one of these queues.

An interrupt is said to have occurred when the count of the element of ***I*** corresponding to an interrupt of that type is increased to reflect that occurrence. At the time of the interrupt the data queue if present would be fed data. In addition to the above a one bit flag is turned on. This requires yet another meta-linguistic state component which we shall designate as ***POLL***. We also add yet another component, ***P***, to our machine which is an integer defining the current level of priority. After interrupts occur they must be detected and serviced, this is accomplished by the following extensions.

The application rule is extended for subrule 11.1.1., where a *lambda-abstraction*, is being applied. If *bv* is conformal with $a_1 \ldots a_n$, and ***POLL*** is in the on condition then:

→ {*()*; **bind**$_2${*bv*; { ***((ehE•lE)•tE)*** | ***((ehE)•E)*** }$_3$; a_1; ...a_n};
    (DISPATCHER) ***•ES•exp-seq•()***; {***S***; ***E***; ***C***; ***D***}}

The action of DISPATCHER is to search ***I*** for an element u;
if u = (*priority-level*$_i$ n_i • k_i) for some u ∈ ***I***,
    and *priority-level*$_i$ >***P*** and n_i>0, is the interrupt element of the highest priority, pending interrupt then u←(*priority-level*$_i$ n_i-*1* • k_i). Should no interrupt of sufficient priority pend ***POLL*** is turned off and control returns from DISPATCHER.
        (note: ← is used to denote "is updated to")
For this highest priority, pending interrupt DISPATCHER will invoke:
((ELT EXTERNAL-EVENTS-CHANNELS k_i) k_i) with ***POLL*** turned off and ***P*** = *priority-level*$_i$.

When and if control returns ***P*** is reset to zero and the search, of ***I***, for the highest is repeated and when no more pend ***POLL*** is turned off. Control then returns from the DISPATCHER.

To repeat what has already been mentioned above: Uncontrolled continuation (applying states) from high priority interrupts could cause loss of sensitivity to lower priority interrupts which were in process and interrupted. In some cases this could be exactly what was intended.

Summing up, it seems that some sort of process data object, a blocked process queue, extensions to the scheduling in DISPATCHER, etc could all be brought together under a unified processing model. The main problem seems to be that no compelling model has appeared and as yet no compelling interest in a better model has developed.

The application rule is extended for subrule 11.3.1. where a conformal *fbpi* is being applied, and ***POLL*** is in the on condition.

$$\rightarrow \{\mathit{()};\ \mathbf{bind}_2\{\mathbf{bv\text{-}of}\{x\};\ \mathit{((ehE)\bullet E)};\ a_1;\ldots a_n\};\ (\text{DISPATCHER})\bullet\ \mathit{lc}_{entry}\{x\};\ \{S;\ E;\ C;\ D\}\}$$

Rule 11.6.1.1., where *funargs* with a *fbpi* as *e-part* are applied, similarly extended (using the environment of the *s-part* of course); as are the macro application rules, 9.1.1., 9.2.1., 9.3.1.1., and 9.3.2.1. .

The Go expression rule 13.6.1 is also extended so that in the case of a pending interrupt:

$$\rightarrow \{S_1;\ E;\ (\text{DISPATCHER})\bullet\ (\{SEQ_1\ /\ SEQ_2\}\ \bullet S_1)\bullet\ (st\text{-}lab_1\ s_2\ldots)\bullet(s_1\ldots)\bullet C_1;\ D\}$$

These extensions allow for the timely service of the events that are pending.

The number of, priority levels of, and types of interrupts of a given ***SECD*** machine are rather fixed and ad hoc. The allocation of new ***I*** and ***R*** require capability not provided as LISP basic operators. System programmer help is required to replace ***I*** and ***R***. A basic primitive for replacing the elements of ***R*** with new allocations of queue space, is provided and is called NEWQUEUE.

(NEWQUEUE k m) Replaces the kth element of ***R*** with a new vector which has capacity for m elements. This operation can only take place when no interrupts are pending. Upon successful completion NEWQUEUE returns the new vector.

EXTERNAL-EVENTS-CHANNELS has a value which is a vector whose kth element is a function of one argument, the integer k, which should be the service function for interrupts of type k. See Table 1, for the detailed definition for each channel. As this vector is a LISP reference vector normal vector operations may be used on it, with caution!

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.

External events may be posted by LISP programs through the use of the basic function POST.

(POST k data...) Causes pending of a user interrupt of class k in *I*. Enqueues data, an integer if present, in ***R***. Returns k if the enqueuing was successful and NIL otherwise.

(EAT k) Eats one data element of ***R***. Returns NIL if the queue is empty or if no such queue is present, otherwise the value is the integer data value eaten.

The ability of the user to redefine EXTERNAL-EVENTS-CHANNELS may lead to inconsistency. It is still an open issue whether this ability is worth its added danger.

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 the explicit details for each channel. The semantics rule should be scrutinized for occurrences of ERR2, that is, instances of error channels being invoked. These distinctions are not a property of the service expression but rather how it is invoked.

```
(ERR2 n) → %(LAMBDA (?ARGS?)
                (S,ERRORLOOP n ?ARGS? (STATE)))
```

<u>S,ERRORLOOP</u> becomes a new understood basic function. The current implementation of which is to put you in the break state supervisor. This supervisor runs in its own state but has the interrupted state passed as a parameter. The first action of the break state supervisor is to ask if it should try to run in the interrupted state. It does this by making strange and wonderful tests, one necessary condition of which is to test some "heaven-box". The programmer can force the break supervisor to run in the "safe" state by causing the heaven-box to be set to zero through some external means. In whatever state it runs it does so by causing the nth element of PROGRAM-EVENTS to be applied to n, the arguments and the current state.

The user should refrain from updating the "safe" state once he is running in it. For PROGRAM-EVENTS variables bound in the user's state, the user may invoke the channels in a similar manner and may update each channel with his own definitions.

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	'UR 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 ' EVAL 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 ... EVAL) 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.

GLOBAL ENVIRONMENTS, EXIT ROUTINES AND AN EFFICIENCY DEVICE

In the original definition of *E* the metalinguistic function: **fluid**{id_i;x;y;E} was not fully explained. The fact is that a search avoidance mechanism is built into the state machinery. This was called the shallow binding by Bobrow and Wegbreit[4].

A provision for general exit functions not unlike that suggested by Bobrow and Wegbreit is also provided. This device is used by the search avoidance scheme. It is therefor convenient to introduce both in this section.

The earlier definition of *E* also had reference to the the case that pertains when the bindings search is applied to ***nilE*** (which defines the end of the environment created by **bind**). The comment was that some agreed upon binding would be produced. This section also implements that notion.

To the normal {*S;E;C;D*} state we add a new component called the exit which we shall denote by *X*. This gives {*S;E;C;D;X*} as the state.

Recall from the previous sections that the state was applied to *M*, that this notation was used to model the notion that the state transitions take place with respect to a memory. For the implementation of search avoidance, a special metalinguistic component is added to denote the current environment-path. Environment-path identifiers are metalinguistic data objects whose principle property is that they identify an environment search path. A secondary, but useful, property is that they possess some space for saving and restoring some state components during path switching.

The metalinguistic function **bind** will be presumed to have been extended so that when it binds the ***ident*** (FLUID *iden*) it also stores a reference to the new fluid binding *(s-exp • ident)*, and the current environment-path identifier. The storage for these two objects is called the shallow-cell. Each *id* which has ever been fluid bound has an associated shallow-cell. This fact should be kept in mind as shallow-cells may very well be a scarce resource. The following diagram details the structure of the shallow-cell in one actual implementation.

sbst, the shallow binding state ptr.
sboff, the offset ptr. to the actual value cell.
acd, the environment path identifier.
sbid, the id-delta, an offset ptr. to a communication cell whose contents is an *id*.

The function **fluid** is designed to avoid the environment search for a free variable in the case that the path identifier (*acd*) of the shallow-cell of the *id* in question, is identical to the current environment-path identifier.

The search may also be avoided in the case that a distinguished *acd* indicates that this *id* has never been fluid bound. In which case the identifier in question could only have been locally bound (in which case we don't care), or it was globally bound in which case the binding is correctly indicated if the *sbst* is EQ to the current global environment path identifier.

In the case that **fluid** is unable to avoid search, the shallow cell is reestablished for the current path.

As a result of much consideration, several false starts, and dogged persistence, the ideal embodyment of environment-path identifiers is believed to be: state descriptors. The total state then consists of the ordinary state, now shown to be {***S;E;C;D;X***}, applied to ***M***, applied to the environment path identifier.
i.e. {***S;E;C;D;X***} {***M***} {*sd*} .

State descriptors (*sd's*) have the following components:

1. The ***D-part*** which is an ordinary state {***S;E;C;D;X***} also known as an activation record. The ordinary state is distinguished because we will often copy it, but we would seldom copy the total state.

2. The path descriptor of the using state, which is the *sd* that identifies the path that is to be restored when control exits this path. For state descriptors that are not "in control" this is a self denotation. The path descriptor of any state is in effect the most recent *sd* with respect to which evaluation takes place.

3. The exit field hideaway, which is the ***X*** of the using state. The use of this field will be explained in the details that follow.

4. The ***gloE-part***, or global environment path identifier which is a ***gloE***. See pages 8 and 9.

A few words of comment about ***gloE***:

First of all, ***gloE*** are not regular data objects, they are however components of *sd* which are data objects.

Also, *glodat* which are ignored by the basic system processes, are inherently dangerous. User defined *glonot* prescriptions that utilize them must maintain consistent interpretations for them whenever *glolsts* are shared.

Meta-syntactically: sd_i = {D_i;sd_j;***X;gloE***}. Remember that *sd's* have no equivalence preserving, external representation, unlike all other LISP data objects.

It is now possible to explain what was meant by the phrase in the definition of ***E{id$_l$}*** :

if ***E = nilE*** then some agreed upon global binding, **global**}*id$_l$*;x;y}.

nilE the distinguished empty environment acts as a terminator for that part of the environment created by **bind**, which shall be referred to as the normal environment. The ***gloE*** of the current path descriptor defines an addition or extent to the normal environment known as the global environment. The definition of the environment function provides for the production of bindings on first reference, global context switching, direct access data bases, and in general is limited only by the imagination of the programmer, and his desire to be consistent.

For what follows, it will be sufficient to just describe the total state as if it were: $\{S;E;C;D;X\}sd$. The following description is given as modifications and additions to several of the existing rules. Most readers will find these modifications complicated and uninteresting and are encouraged to skip to the next section.

2. Value return restoring the former state. With exit functions added.

$\{x\bullet S_1;\ E_1;\ ();\ \{S_2;\ E_2;\ C_2;\ D_2;\ X_2\};\ X_1\}sd_1$

if X_1 is () or X_0 then $\rightarrow \{x\bullet S_2;\ E_2;\ C_2;\ D_2;\ X_2\}\ sd_1$

Comment: () as exit indicates no exit function and stack contiguity. X_0 just indicates no exit function.

if X_1 is $sd_x = \{D_x;\ sd_y;\ X_y;\ gloE_x\}$

$\rightarrow \{x\bullet S_1;\ E_1;\ ();\ \{S_2;\ E_2;\ C_2;\ D_2;\ X_2\}\ ;\ X_y\}\ sd_y$

and $sd_x \leftarrow \{D_x;\ sd_x;\ X_0;\ gloE_x\}$

Note: $\leftarrow$ is used to denote "is updated to be".

Comment: This case illustrates the return of control to a context described by another path descriptor.

if X_1 is a pair (a • b) then

$\rightarrow \{x\bullet S_1;\ E_1;\ \text{'}APP_1{*}S_1\text{'}\bullet()\bullet a\bullet();\ \{S_2;\ E_2;\ C_2;\ D_2;\ X_2\};\ b\}sd_1$

Comment: Illustrates a composition of exit functions.

if X_0 is X_{spoil} then

$\rightarrow \{x\bullet S_2;\ E_2;\ C_2;\ D_2;\ X_2\}\ sd_1$ and, **spoil-fluid{fvd}**.

Comment: X_{spoil} is used to invalidate the shallow cells of fluids bound when this state was created. It will require either a new binding or a subsequent environment search to reestablish the shallow cell. For compiled code, the shallow cell's original contents were saved on function entry and restored on exit. X_{delete} and X_{spoil2} are equivalent to () and X_{spoil} from the point of view of this semantics. They serve to indicate stack frames whose deletion is other than the normal case.

otherwise $\rightarrow \{x\bullet S_1;\ E_1;\ \text{'}APP_1{*}S_1\text{'}\bullet()\bullet X_1\bullet();\ \{S_2;\ E_2;\ C_2;\ D_2;\ X_2\};\ X_0\}sd_1$

Comment: The specified exit function is applied to the value, control will then normally return to former state.

6. Closure evaluation

$\{S; E; funarg \bullet C; D; X_1\} sd_1$

where e is the *e-part* and sd_2 is the *s-part* of *funarg*.

and $sd_2 = \{D_x; w; v; \mathbf{gloE}_x\}$ and

D_x has E_1 as its ***E-part***.

$\rightarrow \{(); ((ehE \bullet lE_1) \bullet tE_1); e \bullet (); \{S; E; C; D; X_1\}; sd_3\} sd_3$

if $w = sd_2$ then $sd_3 = sd_2$; $sd_2 \leftarrow \{D_x; sd_1; X_{decte}; \mathbf{gloE}_x\}$

if $w \neq sd_2$ then $sd_3 = \{D_x; sd_1; X_{decte}; \mathbf{gloE}_x\}_{\mathrm{new}}$

Comment: This expression represents an expression closed with respect to the environment of *sd*. In the absence of updating, such expressions denote the same value regardless of the context in which the closure is evaluated.

9. Macro application (only the significant changes)

$\{y \bullet S; E; MAPP \bullet x \bullet C; D; X_1\} sd_1$

9.1. If x = *mbpi*

9.1.1. If **bv-of{x}** and $a_1 \ldots a_n$ are conformal

$\rightarrow \{(); \mathbf{bind}\{\mathbf{bv\text{-}of}\{x\}; y; ((ehE) \bullet E)\}; lc_{entry}\{x\}; \{S; E; C; D; X_1\}; X_2\} sd_1$

where X_2 is:

() or X_0 in the case where no fluids were bound;
or in the case where fluids were bound, their old *shst*, *shoff*, and acd_1 are pushed on the stack and an exit function internal to the *mbpi* restores them prior to normal contour exit. acd_1 is sd_1 if the old *acd* in the shallow-cell is the same as sd_1 otherwise acd_1 is a spoiler acd. The spoiler acd when restored prevents **fluid** from avoiding the environment search.

9.1.2. Otherwise $\rightarrow D_{non\text{-}conformal\text{-}app}$,

9.2. If x = %(<u>MLAMBDA</u> *bv* • *exp-seq*)

9.2.1. And y is conformal with *bv*

$\rightarrow \{() \bullet (); \mathbf{bind}\{bv; y; ((ehE \bullet ()) \bullet E)\}; STMT_2 \bullet exp\text{-}seq \bullet ();$
$\{S; E; C; D; X_1\}; u\} sd_1$

where u is X_{spoil} or X_{spoil2} if there were any fluids in *bv* otherwise X_{decte} or ().

9.2.2. Otherwise $\rightarrow D_{non\text{-}conformal\text{-}app}$, 9.3. If x is a *macro-funarg* with z the *e-part* and sd_2 the *s-part* where sd_2 = {D_x; w; v; ***gloE)**$_x$*} and D_x has an E_3 as its ***E-part*** then

9.3.1. If z is *mbpi*

9.3.1.1. If **bv-of}z}** and y are conformal

→{*()*;**bind**{**bv-of**{**z**}; y; ***((ehE•lE$_1$)•tE$_1$)*** : ***lc$_{entry}$**{z}*; ***{S;E;C;D;X$_1$};sd$_3$}sd$_3$***

9.3.1.2. Otherwise → ***D$_{macro\text{-}non\text{-}conformal}$***,

9.3.2. If z = %(<u>MLAMBDA</u> *bv* • *exp-seq*)

9.3.2.1 If y is conformal with *bv*

→{()•*()*; **bind**{***bv; y; ((ehE•lE$_1$)•tE$_1$)***}; ***STMT$_2$•exp-seq•()***; ***{S; E; C; D; X$_1$}; sd$_3$}sd$_3$***

where

if w = ***sd$_2$*** then ***sd$_3$*** = ***sd$_2$***;

sd$_2$ ← {***D$_x$; sd$_1$; X$_{\{decte\ |\ spoil2\}}$; gloE$_x$***}

if w ≠ ***sd$_2$*** then ***sd$_3$*** = {***D$_x$; sd$_1$; X$_{\{decte\ |\ spoil2\}}$; gloE$_x$***}$_{new}$

9.3.2.2. Otherwise → ***D$_{macro\text{-}non\text{-}conformal}$***,

.
.
.

Comment: Macro application with respect to an environment causes a change in the current environment-path designator. It is possible to use a sd_2 itself as the current path designator unless sd_2 is already in current use, as indicated the presence of a using state. note: The notational form {***D***; a; b; ***gloE***}$_{new}$ indicates allocation of a new *sd* from the heap.

11. Ordinary application, (significant changes only).

{a_n•...a_1•***S; E; ({APP$_1$ | APP$_2$}$_3$•S)•c•***x•***C; D; X$_1$}sd$_1$***

11.1 If x = ***APP$_3$*** then

{a_n•...a_1•***S; E; ({APP$_1$ | APP$_2$}$_3$•S)•c•APP$_3$•*** *bv* • *exp-seq*•***C$_2$; D***}

11.1.1. If *bv* and a_1 ... a_n are conformal

→{()•*()*; **bind$_2$**{*bv*; { ***((ehE•lE)•tE)*** | ***((ehE)•E)*** }$_3$; a_1; ...a_n}; ***ES$_2$•exp-seq•()***; ***{S$_1$; E; C; D; X$_1$}***; u} ***sd$_1$***

where u is X_{spoil} or X_{spoil2} if there were any fluids in *bv* otherwise X_{decte} or ().

.
.
.

11.4.1.4. If x is the EVAL, and a_1 and a_2 are present.
$\{a_n\bullet\ldots a_1\bullet S;\ E;\ (\{APP_1\ /\ APP_2\}_3\bullet S)\ \bullet()\bullet$<u>EVAL</u>$\bullet C;\ D;\ X_1\}sd_1$

11.4.1.4.1. If a_2 is sd_2
$\rightarrow\{();\ ((ehE)\bullet E_1);\ a_1\bullet();\ \{S;\ E;\ C;\ D;\ X_1\};\ sd_3\}sd_1$
where if $a_2 = sd_2$ where $sd_2 = \{D_x;\ w;\ v;\ gloE_x\}$ and
D_x has an E_1 as its ***E-part*** then
if $w = sd_2$ then $sd_3 = sd_2;\ sd_2 \leftarrow \{D_x;\ sd_1;\ X_{decte};\ gloE_x\}$
if $w \neq sd_2$ then $sd_3 = \{D_x;\ sd_1;\ X_{decte};\ gloE_x\}_{new}$

.
.
.

11.4.2.1. <u>CALL</u>, the function to apply the operator which is computed after the operands to however many arguments were transmitted.
$\{a_n\bullet a_{n-1}\bullet\ldots a_1\bullet S;\ E;\ (\{APP_1\ /\ APP_2\}_3\bullet S)\ \bullet()\bullet$<u>CALL</u>$\bullet C;\ D;\ X_1\}sd_1$

.
.
.

11.4.2.1.2. If a_n = %(<u>FUNARG</u> y $sd_2\bullet$ z) where $sd_2 = \{D_x;\ w;\ v;\ gloE_x\}$ and D_x has an E_3 as its ***E-part*** then
11.4.2.1.2.1. If y is *fbpi*.
11.4.2.1.2.1.1. If **bv-of**$\{fbpi\}$ and $a_1 \ldots a_{n-1}$ are conformal.
$\rightarrow\{();\ \mathbf{bind}_2\{\mathbf{bv\text{-}of}\{fbpi\};\ ((ehE\bullet lE_1)\bullet tE_1);\ a_1;\ \ldots a_{n-1}\};\ lc_{entry}\{fbpi\};$
$\{S;\ E;\ C;\ D;\ X_1\};\ sd_3\}sd_3$
where
if $w = sd_2$ then $sd_3 = sd_2;\ sd_2 \leftarrow \{D_x;\ sd_1;\ X_{decte};\ gloE_x\}$
if $w \neq sd_2$ then $sd_3 = \{D_x;\ sd_1;\ X_{decte};\ gloE_x\}_{new}$
11.4.2.1.2.1.2. Otherwise $\rightarrow D_{non\text{-}conformal\text{-}app}$.
11.4.2.1.2.2. Otherwise
$\rightarrow\{a_{n-1}\bullet\ldots a_1\bullet();\ ((ehE\bullet lE_1)\bullet tE_1);\ (APP_2\bullet())\bullet()\bullet a_n\bullet();$
$\{S;\ E;\ C;\ D;\ X_1\};\ ()\}sd_1$
11.4.2.1.3. Otherwise
$\rightarrow\{a_{n-1}\bullet\ldots a_1\bullet();\ ((ehE\bullet lE_1)\bullet tE_1);\ (APP_2\bullet())\bullet()\bullet a_n\bullet();\ \{S;\ E;\ C;\ D;\ X_1\};$
$()\}sd_1$

Comment: This rule is ordinary calling in the current environment with arguments transmitted on the stack.

11.4.2.2. If x is $\underline{\text{STATE}}$ the state saving operator it creates continuations

$\{a_n\bullet\ldots a_1\bullet S;\ E;\ (\{APP_1 \mid APP_2\}_3\bullet S)\bullet()\bullet\underline{\text{STATE}}\bullet C;\ D;\ X_1\}sd_1$

$\rightarrow\{sd_2\bullet S;\ E;\ C;\ D\};\ X_1\}sd_1$

where $sd_2 = \{\{S;\ E;\ C;\ D;\ X\}_{\text{old}};\ sd_2;\ ();\ w\}_{\text{new}}$

and $w = (glonot_1 \bullet glolst_1)_{\text{new}}$ if $(z\ldots\bullet S) = S$, or

$w = (glonot \bullet glolst_1)_{\text{new}}$ if $(z\ldots\bullet S) = (glonot\bullet S)$, or

$w = (glonot \bullet glolst)_{\text{new}}$ if $(z\ldots\bullet S) = (x\ldots\bullet glolst \bullet glonot \bullet S)$,

where $sd_1 = \{D;\ u;\ v;\ (glonot_1 \bullet glolst_1)_1\}$.

Comment: This *sd* constructing operator can only capture the current ***S,E,C***, and ***D***. It can however have a different global environment associated with ***E***. The new global environment is specified by the two optional parameters of STATE.

.

.

.

11.4.7. Closure forming expression.

$\{a_n\bullet\ldots a_1\bullet S;\ E;\ (\{APP_1 \mid APP_2\}_3\bullet S)\bullet()\bullet x\bullet C;\ D;\ X_1\}sd_1$

where x is $\{\underline{\text{FUNCTION}} \mid \underline{\text{LAMBDA}} \mid \underline{\text{MLAMBDA}} \mid \underline{\text{FR*CODE}}\}_2$

and $a_1 = (y \bullet w)$ then

11.4.7.1. If x is FUNCTION or FR*CODE and y is *funarg*$_1$ then

$\rightarrow \{funarg_1\bullet S;\ E;\ C;\ D;\ X_1\}sd_1$

11.4.7.2. Otherwise, $\rightarrow \{funarg_2\bullet S;\ E;\ C;\ D;\ X_1\}sd_1$.

where *funarg*$_2$ is created and has u as *e-part* and *sd*$_2$ as *s-part*

where u is:

$\{\ y \mid$

$\%(\underline{\text{LAMBDA}}\ y \bullet w) \mid$

$\%(\underline{\text{MLAMBDA}}\ y \bullet w) \mid$

$\%(\underline{\text{FR*CODE}}\ y \bullet w)\ \}_2$,

where $sd_2 = \{\{();\ E;\ ();\ ();()\}_{\text{old}};\ sd_2;\ X_0;\ \mathbf{gloE}_1\}_{\text{new}}$ and $\mathbf{gloE}_1$ is the $\mathbf{gloE}$ of sd_1.

i.e., the lexical bindings (if present) are operative in the environment of the closure.

.

.

.

11.6. If x = *funarg* and y is the *e-part* and sd_2 the *s-part*,
where $sd_2 = \{D_x;w;v;\mathbf{gloE}_x\}$ and D_x has an E_1 as its ***E-part*** then

11.6.1. If y is an *fbpi*.

11.6.1.1. If **bv-of}y}** and $a_1 \ldots a_n$ are conformal then
$\rightarrow\{();\ \mathbf{bind_2}\{\mathbf{bv\text{-}of}\{y\};\ ((ehE \bullet lE_1)\bullet tE_1);\ a_1;\ \ldots a_n\};\ lc_{entry}\{y\};\ \{S;\ E;\ C;\ D;\ X_1\};\ sd_3\}sd_3$

11.6.1.2. Otherwise $\rightarrow D_{non\text{-}conformal\text{-}app}$.

11.6.2. Otherwise $\rightarrow\{a_n\bullet\ldots a_1\bullet();\ E_1;\ (APP_1\bullet())\bullet()\bullet y\bullet();\ \{S;\ E;\ C;\ D;\ X_1\};\ sd_3\}sd_3$
where
if $w = sd_2$ then $sd_3 = sd_2$; $sd_2 \leftarrow \{D_x;\ sd_1;\ X_{decte};\ \mathbf{gloE}_x\}$
if $w \neq sd_2$ then $sd_3 = \{D_x;\ sd_1;\ X_{decte};\ \mathbf{gloE}_x\}_{\mathbf{new}}$

11.7 If x is $sd_2 \rightarrow \{a_1\bullet S_2;\ E_2;\ C_2;\ D_2;\ X_2\}sd_3$
where $sd_2 = \{D_x = \{S_2;\ E_2;\ C_2;\ D_2;\ X_2\};\ w;\ v;\ \mathbf{gloE}_x\}$

where
$sd_3 = \{D_{nilE};\ w;\ v;\ \mathbf{gloE}_x\}_{\mathbf{new}}$

Comment: A fact that this meta-language may not adequately convey is that S_2 and C_2 of the *sd* are copies but E_2 and D_2 are shared references.

.

.

.

11.4.2.?. The EXFN function. Not previously defined.

$\{z\ldots x\bullet S;\ E;\ (APP_{\{1\,|\,2\}}{*S})\bullet()\bullet\underline{\text{EXFN}}\bullet C;\ D;\ X_1\}\ sd_1$

$\rightarrow\{x\bullet S;\ E;\ C;\ D;\ X_2\}sd_1$
if $X_1 = sd_2 = \{D_2;\ a;\ b;\ \mathbf{gloE}_2\}$ then $X_2 = sd_2 \leftarrow \{D_2;a;c;\mathbf{gloE}_2\}$,
if $b = \{()\ |\ X_0\}$ then $c = x$,
otherwise $c = (x \bullet b)$
if $X_1 = ()$ then $X_2 = (x \bullet X_0)$,
otherwise $X_2 = (x \bullet X_1)$.

Comment: This addition to the exit routine capabilities could be used to establish a user defined exit function.

13.12. Meta application of a *context-closure*

$\{a_1\bullet S;\ E;\ SF\bullet\%(\underline{\text{FUNARG}}\ \%(\underline{\text{MU}}\ bv \bullet valuelist) \bullet sd_2) \bullet C;\ D;\ X_1\}sd_1$
$\rightarrow\{v_n\bullet\ldots v_1\bullet S;\ E_1;\ (APP_1\bullet())\bullet()\bullet APP_3\bullet bv\bullet\ a_1\bullet();\ \{S;\ E;\ C;\ D;\ X_1\};sd_3\}sd_3$
where *valuelist* = $(v_1\ldots v_n)$ and sd_2 has E_1 as its ***E-part***.
where $sd_2 = \{D_x;\ w;\ v;\ \mathbf{gloE}_x\}$ and
D_x has an E_1 as its ***E-part*** then

if w $= sd_2$ then $sd_3 = sd_2$; $sd_2 \leftarrow \{\boldsymbol{D_x}; sd_l; \boldsymbol{X_{delete}}; \boldsymbol{gloE_x}\}$
if w $\neq sd_2$ then $sd_3 = \{\boldsymbol{D_x}; sd_l; \boldsymbol{X_{delete}}; \boldsymbol{gloE_x}\}_{\text{new}}$

THE LISP/370 DESTRUCTIVE STREAM FACILITY

In Stoy and Strachey[1] *streams* were used as vehicles for the transfer of information in systems. Their streams were destructive, i.e., a *destructive stream* is a stream with updatable private storage. This allows successive items of a stream to occupy the same storage. Burge[2] has described an even more general stream model in which *destructive streams* are a special case. In Burge's more general model non-destructive streams are applicable functions, which are retained and thus reuseable. "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."

Landin [5] appears to have first proposed streams as an alternative to lists. He used streams to model the concept of a "control-list", a term he used to mean the successive values of the for-statement control variable. He noted streams similarity to coroutines and suggested them as a model for input/output.

In LISP/370 we copy the Stoy and Strachey destructive stream concept to a large extent. Functional streams are definable, but we choose to supply with the language a data-structure model for streams and basic facilities to manipulate it. The data-structure model was selected instead of functionals after considering current efficiency tradeoffs.

The non-empty stream data structure has two components; the first element is the current item at the *head* of the sequence, and the second element defines the *rest* of the sequence, a (possibly empty) stream which is denoted by either a stream terminator structure or another pair. An empty stream is denoted by a stream terminator, which is either an *ssd* (for special stream descriptor) or *other-atom* (any other non-pair data structure). The *rest* is not an accessible component; the stream it denotes is computed by one of the stream successor functions.

It can be seen from the following description that several types of streams are provided. We intend that certain functions which scan or create lists can be converted into functions which scan or create streams. The resulting functions will benefit from the more abstract nature of streams; for example, the elements of a stream need not all exist in storage but can be generated as needed. Because the stream facilities provided are destructive of the stream they use, we cannot conveniently convert most list functions, which are often expected to be retentive.

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 for Teaching Programming Linguistics." *Proc. 23rd ACM Conf.*, 395 (1968).
5. P. J. Landin. "Correspondence Between ALGOL 60 and Church's Lambda-notation." *Commun. ACM* **8**, Part 1, 89, Part 2 158 (1965).

A stream data structure, *stream* is
  either a pair (*heads* • *rests*) where,
    heads is the next item of the sequence (any *s-exp*),
    and *rests* is a *stream* which defines the rest of the sequence.
  or an atom *strmterm*.

A *strmterm* may be a special stream descriptor *ssd*, (implemented by a vector), or any other non-pair *other-atom*. An *ssd* 'implies' a function which defines another stream (probably occupying the same pair as the original stream). *Fast-streams* of characters are an efficient subset of such implicitly defined stream functions.

An *ssd* has the form <*rfn bd asc* [*any*...]*s-type*>,
  where *rfn* is $\{next \mid write \mid bidirect \mid \ldots\}_1$†
    where *next* is a one argument, input-stream successor function,
      next: $STREAM \sim OUS \sim STMTERM \Rightarrow STREAM$,‡
    and *write* is a two argument, output-stream successor function,
      write: $S\text{-}EXP \times STREAM \Rightarrow STREAM$,
    and *bidirect* is a three argument, bi-directional stream successor function,
      bidirect: $S\text{-}EXP \times BDS \times \{\text{IN} \mid \text{OUT}\} \Rightarrow STREAM$,
  and *bd* is the buffer description:
    nil in the case of slow-streams,
    or a *fast-stream-buffer* in the case of fast-streams.
  and *asc* is the *associated-states*, which is an *a-list*,
  and *any* is any stream dependent information that the user provides,
  and *s-type* the stream type should conform to *rfn*, i.e.
    $\{\text{IN} \mid \text{OUT} \mid \text{BDS} \mid \ldots\}_1$.

Thus it can be seen from the structure description that streams are differentiated as to input-streams *ins*, output-streams *ous* and bi-directional streams *bds*.

$STREAM = LISTS \cup INS \cup OUS \cup BDS$
  and **ins** $\in$ *INS* is (*heads* • < *next* x... IN>),
  and **ous** $\in$ *OUS* is (*heads* • < *write* x... OUT>),
  and **bds** $\in$ *BDS* is (*heads* • < *bidirect* x... BDS>)

A fast-stream-buffer has the form:
  < *buffer begindex curindex endindex* x...>,
    where *buffer* is *nil* for inactive streams,
      and *string* otherwise,
    where *begindex* the beginning index is a {0 | 1 | ... size{*string*}},
    and *curindex* the current character index is a {0 | 1 | ... size{*string*}},
    and *endindex* the boundary index is a {0 | 1 | ... size{*string*}}.

† Subscripts are used to indicate a correspondence.
‡ ~ is a left-associative set difference operator.

The basic functions on streams are: HEADS, NEXT, WRITE, IS-EOB, DEF-STRM, and NULLS.

The following axioms characterize system provided streams:

For s $\in$ {*STREAM* ~ *OUS*};

(NEXT (WRITE x s)) = s,
If (NOT (NULLS s)) then (WRITE (HEADS s) (NEXT s)) = s,
(HEADS (WRITE x **stream**)) = x,
(NEXT **ous**) is a domain error,
(HEADS **strmterm**) is a domain error,
(NEXT **strmterm**) is a domain error,
(NOT (NULLS (WRITE x **stream**))),
(NULLS **strmterm**),
If (EQ (HEADS **stream**) **stream**) then (NULLS **stream**).

Not only are streams similar to lists; lists may be used as streams, *liststrm*, which is any non-empty stream that is not included in:

INS ∪ *OUS* ∪ *BDS* .

Keep in mind that the argument list is updated by most of the stream primitives. It should be noted that we often think of lists as scanned from the left, and also prefer to augment on the left. The basic stream functions provided reflect this bias when operating on *liststrm*. This is to be contrasted with current preferences for input and output files. Input files are also scanned from the left, but output files are usually augmented on the right. Streams can be so defined so that they are scanned or augmented in either direction. It is important that the user keep the conventions of stream producers and stream consumers consistent.

The list analogy is show in the following chart.

Stream Operator	**Non-Destructive List Analogy**	**Destructive *Liststrm* Analogy**
(HEADS x)	(CAR x)	(CAR x)
(NEXT x)	(CDR x)	(RPLACD (RPLACA x (CADR x))(CDDR x))
(WRITE x y)	(CONS x y)	(RPLACA (RPLACD y (CONS (CAR y)(CDR y)))x)
(NULLS x)	(ATOM x)	(OR (ATOM x)(EQ (CAR x) x))

Slow-streams provide for the generality of user defined streams. Each successor stream is defined by the application of the user provided *rfn*. These functions need not comply with the destructive stream analogy. Nevertheless, what LISP/370 provides is primarily meant to exhibit the sequential evaluation behavior of destructive streams.

Fast-streams provide a more efficient successor method through the use of *buffers*. In LISP/370 this is only provided for character-object streams. System dependent input and output is achieved through the use some distinguished fast-streams, or through user defined system dependent fast-streams. NEXT and WRITE have been extended to give fast-streams special treatment. Fast-streams for real input-output files contain the essential file information in their *ssd*.

As yet the system provides no bi-directional stream facilities; they are included in this description as a suggestion for user development.

The full descriptions of the basic stream primitives follow:

(NULLS **stream**)

This tests if **stream** is an empty stream. Returns **true** if **stream** ϵ *STMTERM* or if *heads* is the stream itself; it returns **false** otherwise.‡ Defines the set of empty streams, *NULLS*. There may be many empty streams that are not EQUAL.

Consider the following fast-stream which is not empty but is nearly so: %L1†=(***eob*** • %L2=<*next* <*string* 0 n n> *nil* IN>) where a subsequent application of NEXT will produce %L2 as value and %L1→ %L1=(%L1 • %L2), an empty stream.

The interpretation of the original stream is as a stream with end-of-block ***eob*** (represented by %.EOB) as its last and only item. The stream itself serves as an emptied stream indicator when it appears as *heads*. We install this convention because many routines depend upon the EQ'ness preservation property of destructive streams.

(HEADS s) where s $\notin$ *STMTERM*

HEADS is the access function used to peep at the current element of the stream. It has no side effects and can be used repeatedly without advancing the sequence.

heads{%L1=(*heads* • *rests*)} →
    if *heads*≠%L1 then *heads*,
    if *heads*=%L1 then accessing empty stream error.

‡ In LISP/370 **false** is denoted by the distinguished object *nil* and **true** by any other object.

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

(NEXT s) where s ϵ *STREAM ~ OUS ~ STMTERM.*

NEXT is a function from streams to streams. For most arguments it 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-atom*) } →
other-atom and %L1 → %L1=(%L1 • *other-atom*).

next{ %L1=(x y • z)} → %L1=(y • z) .

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

next{%L1=(*heads* • <*bidirect nil* x... BDS>)} → *bidirect*{*nil*;%L1;IN}.

```
next{%L1=(heads • %L2=<rfn %L3 x... s-type>)}
   where %L3=<buffer begindex curindex endindex>
         Note: By convention, when buffer is nil then curindex must equal
         endindex and the rfn must replace it with a string which it must allocate.

   if curindex ≥ endindex then
      if heads = eob then
         if %L2 = <next ... IN> → next{%L1}

         if %L2 = <bidirect ... BDS> → bidirect{nil;%L1;IN}

      if heads ≠ eob → %L1=(eob • %L2),
       (This illustrates the production of the end-of-block symbol after the last
       data)

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

(WRITE s-exp stream)

write{x;*atom*} → (x • *atom*) .

write{x;%L1=(y • z)} → %L1=(x y • z)
where z = *other-atom* or <*next* w... IN> or (*heads* • *rests*).
WRITE to an input stream is was called PUTBACK by Stoy and Strachey.

write{x;%L1=(y • <*write nil* w... OUT>)} → *write*{x;%L1}.

write{x;%L1=(y • <*bidirect nil* z... BDS>)} → *bidirect*{x;%L1;OUT}.

write{x;%L1=(y • %L2=<*rfn* %L3 z...>)}
where %L3=<*string begindex curindex endindex*>, and *rfn* ≠ <*next* ... IN>
if *curindex* < *endindex*, and x is a *character*,
→ %L1=(x • %L2=<*rfn* %L3 z...>)
and %L3 →
%L3=<**storechr**{*string;curindex;x*} *begindex curindex+1 endindex*>
(Notice that WRITE to a fast-stream does augmenting on the right.)

otherwise → *rfn*{x;%L1;OUT}.

(TEREAD stream)

TEREAD repeatedly nexts the current stream until it encounters an end-of-block condition and leaves the stream in an end-of-block condition. Primarily intended for input fast-streams.

teread{*atom*} → (***eob*** • *atom*).

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

For x ≠ ***eob*** :
teread{%L1=(x • *other-atom*)} → %L1=(***eob*** • *other-atom*).

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

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

teread{%L1=(x[illegible] w...>)}- [illegible]
→ %L1=(***eob*** • <*rfn fast-stream-buffer*, w...>)

(TERPRI **stream**)

TERPRI forces the stream-dependent successor function if one is present.

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

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

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

terprix{x; %L1=(y • <*rfn* z...>)} → *rfn*{x;%L1;OUT}.
(Notice that the *rfn* has ***eob*** as the object written. This convention serves as a signal that TERPRI is happening.)

(IS-EOB **x**)

Predicate that returns ***true*** if the argument value is the ***eob*** distinguished object and () otherwise.

(DEF-STRM **heads rests**)

Creates the new stream:

(*heads* • *rests*)
where *heads* is the value of **heads**,
and *rests* is the value of **rests**.

Some Distinguished Streams

LISPIT the console input-stream.

LISPIT is a fluid variable with the following initial value:
%L1=(***eob*** • <LISPITTIN < *nil* 0 0 0> *asc nil* IN>)
where *asc*=((DEVICE • CONSOLE)(MODE • I)(QUAL • V)(OWN • %L1)).

After the file is activated:

%L1=(*heads* • <LISPITTIN <*string begindex curindex endindex*> *asc p-list* IN>)
where *p-list* denotes a system dependent I/O control block, or *nil*
and LISPITTIN denotes a console input-stream successor function which is capable of activating the file when the *p-list* field contains *nil*. More precisely, LISPITTIN has such a function as the value of its binding in the initial global-environment.

The function LISPITTIN 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/370 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 *begindex* and *curindex* to 0, and *endindex* to linelength.

4. Applying LISPITTIN to the now active stream.

When LISPITTIN is applied to an active stream it causes a system dependent console input operation to refill *string*, resetting the contents length of *string* to the actual number of characters read, setting *endindex* to that number also, and setting *begindex* to zero and *curindex* to one. If the number of characters read was zero the stream becomes:

%L1=(***eob*** • <LISPITTIN <'' 0 0 0> *asc p-list* IN>).

When more than zero characters were read it becomes:
%L1=(c_0 • <LISPITTIN <'$c_0 \ldots c_{endindex-1}$' 0 1 *endindex*> *asc p-list* IN>).

LISPOT the console output-stream.

LISPOT is a fluid variable with the following initial value:
%L1=(***eob*** • <LISPOTOUT <*nil* 0 0 0> asc_1 *p-list* OUT>)
where asc_1=((DEVICE • CONSOLE)(MODE • O)(OWN • %L1)), and
p-list=nil,
and LISPOTOUT is similar to LISPITTIN 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 *endindex*> *asc p-list* OUT>)
where *endindex* is the system dependent preferred console output line-length and *string* is 'c' . The capacity of *string* is *endindex* characters.

One peculiarity of LISPOTOUT (and hopefully any output-stream which is inactive) occurs when the initial write is in effect a TERPRI.

write{***eob***;%L1=(***eob*** • <LISPOTOUT <*nil* 0 0 0> *asc nil* OUT>)}
→ %L1=(***eob*** • <LISPOTOUT <*string* 0 0 *endindex*> *asc nil* OUT>)
where *string*= '' but has capacity for '*endindex*' 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/370 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-component...)}) |
    (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} or.

(OWN • pair)
```

The value of the FILE property is a list of identifiers corresponding to the naming conventions of the underlying operating system.

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 capacity 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* capacity and *endindex*.

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)(OWN • %L1=*pair*)).

  → %L1=(***eob*** • <,FILEIN <*nil* 0 72 72> asc *nil* 1 IN>)

This defines an input-stream from the file system. The number 72 is the user's idea of the length of the longest record. For most operating systems the actual file characteristics will take precedence. If the file had a maximum record of 120 characters and the first record was 100 characters then the following holds:

next{%L1=(***eob*** • <,FILEIN <*nil* 0 72 72> *asc nil* 1 IN>)}
→ %L1=(c_0 • <,FILEIN <%120'$c_0 \ldots c_{99}$' 0 1 100> *asc p-list* 2 IN>)

  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≥endindex*, *heads* is the ***eob***, and the block read is not empty.

If the first block were empty:
next{%L1}→%L1=(***eob*** • %L2=<,FILEIN <%120'' 0 0 0> *asc p-list* 2 IN>)
and similarly for subsequent empty lines;

on end of file: **next**{%L1}=%L2 and %L1=(%L1 • %L2).

(DEFIOSTREAM asc 72 1)
  where asc=((FILE YYY LIST)(RECFM • V)(MODE • O))

  → (***eob*** • <,FILEOUT <%72'' 0 0 72> asc *nil* 1 OUT>) where '' has capacity for 72 characters.

This defines a file system output-stream. In the case that an old output file exists, we currently update it starting from the position specified. The longest block that we wish to write is 72 characters.

write{c_0;%L1=(***eob*** • <,FILEOUT <'' 0 0 72> asc *nil* 1 OUT>)}
→ %L1=(c_0 • <,FILEOUT <'c_0' 0 1 72> asc *p-list* 1 OUT>)

However,
write{***eob***;%L1=(***eob*** • <,FILEOUT <'' 0 0 72> asc *nil* 1 OUT>)}
→ %L1=(***eob*** • <,FILEOUT <'' 0 0 72> asc *p-list* 2 OUT>)
  An empty record was written.

Summary and Comment

The destructive characteristics of NEXT and WRITE in all but the slow stream case, coupled with the dependence of LISP/370's READ and PRINT functions on this behavior, more or less dictate that user defined slow-streams also conform to the convention that the EQ-uality of the stream be preserved. If the user intends to use non-destructive streams, he cannot expect to substitute them for destructive ones.

Associated with real I/O streams are certain operators that test or change various system dependent status properties, e.g. IOSTATE, IOSTATEW, IS-CONSOLE, and SHUT.

In addition to the queuing disciplines so far discussed LISP/370 has functions for key-sequenced streams. These random access streams are described elsewhere under the descriptions of RDEFIOSTREAM, RREAD, RSHUT, and RWRITE.

Ideally streams would be typed as to queuing discipline and the destructive or non-destructive property, and domain errors would be generated when streams of the wrong type are supplied.

In LISP/370 input streams and lists with FIFO discipline are well provided for, output streams and key sequence streams are of a limited nature, and no bi-directional stream facilities are provided.

In our model streams are similar to lists and obey similar axioms. The user who is familiar with list-processing should have little difficulty using streams and extending his processes into the domain of real I/O.

PART 2

DATA TYPES, POINTERS, VALUES AND PRIMITIVE OPERATORS

It is common, when speaking of LISP data objects, to talk about a vector, or an identifier, or perhaps a list cell, when in fact the object being discussed is actually a *pointer* to that vector, identifier, et cetera. It is useful to form subclasses of this one type (or typeless system). For instance, we have a predisposition to think about numbers, notwithstanding the fact that the object is implemented as a *pointer* of that particular subclass. Our practices and prejudices for such type systems are often varied. In this LISP system a rich set of types has been provided. This multiplicity of types can be either comfort or confusion to the user. It should be noted that the types discussed are a rather ad hoc set of representation types and not abstract data types. Good programming style should dictate the avoidance of representation dependencies. Non-the-less, pragmatics dictate that the abstract data structures be mapped onto the supported types of the underlying representation. This document generally deals with the pragmatics of the underlying system as its principle concern.

In order to prevent confusion, the tenets of the type scheme must be understood by the user. He must know what types are available and have some idea of the useful properties of each. He must know the type specific or generic operators that are available, and what constraints they have, and what useful purpose they serve. He may avail himself of the benefits of static checking though the use of constrained variables. He may constrain both the domains and ranges of functions he defines, thus extending the pragmatics of static type checking beyond the range of the system provided primitives. He cannot (as yet) define his own abstract types nor can he define new representation types.

In this LISP the user may constrain the definitions that he creates but is not required to do so. Many of the system primitives are constrained. Some are not constrained by type but do a considerable amount of internal checking that can lead to a programmed invocation of an ERROR state.

It is intended that the system primitives are implemented in such a way that they do not give the user the capability to destroy the integrity of the implementation. An exception to this is provided in those implementations that support LAP and the compilation of operator-code-abstractions. (a dubious activity for the privileged class of user) It is furthermore the goal that for a suitably constrained program the compiler (or some other preprocessor) may frequently report that no call to ERROR can be evoked by its use. This does not preclude inadequate programs, non-terminating programs, or even programs that indeed can invoke ERROR, but it does serve to assure the user that the particular program is at least "well formed with regard to type". In the presence of sufficient constraints static type checking is utilized by the compiler to produce code that is both efficient and free from static type ERRORs.

The evaluation processes detect and report on violations of the type constraints. Through out this system we maintain this capability for dynamic type checking. A goal of compilation is to satisfy these constraints at compile time, thus removing the necessity for some of the dynamic checks.

The following is a hierarchical classification schema for the computational data types of LISP1.8+0.3i. It serves as a table of contents for the sections that follow.

Each type is given with a notational shorthand called its type class designator in uppercase italic.
† is used to indicate not yet defined or available.
‡ is used to indicate the presence of a limitation of the current implementation.

THE TYPE SCHEMA

Type:

- *Pointer* $\in$ *PTR* :
  - *Simple-objects* $\in$ *SIMP* :
    - *nil* $\in$ *NIL.* (the distinguished object)
    - *Decimal-number* $\in$ *NUM* :
      - *Integer* $\in$ *I* :
        - *small-integer* $\in$ *SMI* ;
        - *intermediate-integer* $\in$ *II* .
        - *large-integer* $\in$ *L* .
      - *floating-point* $\in$ *FP* ;
      - *rational* $\in$ † ;
      - *complex* $\in$ † ;
      - *interval* $\in$ † .
    - *characters* $\in$ *CHAR*;
    - *Truth-values* $\in$ *TVAL*‡.
  - *Composite-objects* $\in$ *COMP*: (has components)
    - *Arrays* $\in$ *A*: (components of uniform type)
      - *pair* $\in$ $PR = PTR^2$; (accessed by unique selectors)
      - *Ranked-arrays* $\in$ *RA*: (uniformly accessed)
        - *Rank-one-arrays* $\in$ *R1A*: (n elements)
          - *Vector* $\in$ *VEC* : (n is fixed)
            - *pointer-vector* $\in$ $VP = PTR^n$;
            - *intermediate-integer-vector* $\in$ $VII = II^n$;
            - *floating-point-vector* $\in$ $VF = FP^n$.
          - *String* $\in$ *STR* : ($n \leq$ *capacity*)
            - *character-string* $\in$ $SC = CHAR^n$;
            - *bit-string* $\in$ $SB = TVAL^n$.
          - *Lists* $\in$ $LIST = PTR^n$: (n variable) ‡
        - *Higher-rank-arrays* $\in$ *MATRIX*† ;
    - *Records* $\in$ *REC.* (components possibly not of uniform type.)
      - *Id* $\in$ *ID* ; (has print name component)
        - *norid* $\in$ *NORID* ;

gensym ∈ *GENSYM* ;
unid ∈ † ;
mobid ∈ † .
Applicative-objects ∈ *APPL*: (system constructs, diverse accessing†)
Abstractions ∈ ‡ *ABST*:
lambda-abstraction ∈ *LAM* ;
mlambda-abstraction ∈ *MLAM* ;
mu-abstraction ∈ *MU* ;
operator-code-abstractions ∈ *ORCD* .
state-descriptor ∈ *SD* .
funargs ∈ *FUN* .
bpi ∈ ‡ :
fbpi ∈ *FBPI* ;
mbpi ∈ *MBPI* .
sf ∈ *SF* :
ur ∈ *UR* = :
fix-ur ∈ *FUR* ;
mult-ur ∈ *MUR* .
Ntuples ∈ *NT*; (uniform accessing, user definable) †
Complex ∈ *PLEX*. (variable set of possibly not uniform component types, user definable, uniform access.) †

POINTERS

LISP/370† implementation, pointer format:

XXXX	YYYY	AAAAAAAAAAAAAAAAAAAAAAAA

The pointers used by LISP/370† are full words (32 bits) and are rich pointers. The first four bits (XXXX) is mandatory type information, the second field (YYYY) is either additional type information or part of the immediate data, the third field is either the address or more immediate data. The reason for having these rich pointers, which do consume more storage space than would otherwise be necessary, has to do with efficiency. Many of the frequently occurring LISP operations require arguments of a specified type. Since the result of an operation performed on an invalid type of argument may actually destroy the LISP system, checking the types of arguments is vital, and this checking may be more efficiently performed if the type code is part of the pointer. This is not to say that other implementation strategies are inefficient. We do like to note that the type cannot in general simply be associated with an address because some types do not denote objects in storage therefore have no address!

Pointers are the principle internal value objects of LISP/370†, all other data types are subtypes of these computational objects, or unions of subtypes. The user will prefer to think of these objects as lists, trees, graphs, etc., but these interpretations are in the mind of the user. The only variables in LISP/370† are those which hold pointer values.

There is a distinction to be made between pointers which contain the address of stored data, and pointers which might be thought of as containing immediate data. In the latter case, the type code in the pointer indicates that the value of this data object is stored in the pointer itself, not in some other storage location. For example, small-integer numbers are stored as part of a pointer with an appropriate type code, while floating point numbers are always stored in a memory location whose address is part of a pointer with appropriate type code.

The reader may reasonably ask why we don't simply call these objects types and avoid the confusion of thinking about pointers that don't point to anything? One answer is that in our early experience they did and we are so conservative that we now view the non-stored objects as existing in some extension to the memory that is neither accessed nor updated. It can be argued that pointers capture the notion that we are representing types which are infinite on a computer with finite limitations.

The significance of this distinction between immediate data and stored data affects the concepts of sharing and updating. Some classes of stored data may be updated, and if shared by several structures, the updated data will also be shared (that is, all of the sharing structures are simultaneously updated). Immediate data is intrinsically non-sharable; therefore, in this sense it is not updatable.

In order to model operations on the storable values, we postulate the existence of several domains:

The basic domains:

PTR = **pointers** = location handles

s-exp = storable values pointed to or denoted by pointers

M = memory

Abstractly, the nature of pointers and memories can be characterized by specifying an initial memory and a few primitive functions for accessing, updating, allocating, freeing, reading into, and writing out from. As will be seen by what follows, LISP/370† supplies such primitives for each primitive subtype.

Because the pointers give an indirect access to storable data types, they are not a primitive type, in the sense that the pointer type is the union of these other types.

Pointer type class: *PTR* = { *SIMP* ∪ *COMP* }

These subclasses will be defined in the following text.

Abstract syntax: *PTR{M}* = **s-exp** = { **simple-object** ∪ **composite-object**}

Pointers (*ptr* ϵ *PTR*) of the memory (*M*) are said to denote **s-exp** a class of values. If $ptr_1\{M_1\} = s\text{-}exp_1$, and $ptr_1\{M_2\} = s\text{-}exp_2$ then it does not follow that: $s\text{-}exp_1 = s\text{-}exp_2$. In other words the value interpretation we give a pointer depends on the current state of the memory.

Primitives:

As this is the typeless or general class and the one obtained by default, many system provided functions are of this domain. In order to avoid an unbounded enumeration, only certain functions and classes of these general functions will be discussed.

The pointer identity predicate: %•EQ : $PTR \times PTR \rightarrow TVAL$
where *TVAL* the domain of truth values is: $\{() \mid \{PTR \sim ()\}\}$

Here we establish the convention for truth values: () for false, any other *PTR* for true. We can see that the set *TVAL* is not supported as a distinct representation type.

(%•EQ ptr_1 ptr_2) = () iff $ptr_1 \neq ptr_2$
otherwise $ptr_3 \in \{TVAL \sim ()\}$

Notation: The use of ƀ to indicate required spaces is eschewed here and in much of what follows. The ordinary space is thought to suffice.

Comment: Identity in the sense of being the same pointer.

The EQ relation is of singular importance because it defines the separate elements of *PTR*, the EQ-class objects.

The allocation functions:
To allocate a *PTR* one must allocate some underlying primitive subtype object. These operators (to be described) will allocate a new pointer denoting a described object of that subtype.

The type predicates:
%•*type-class* : $PTR \rightarrow TVAL$
For all elements of *type-class* except NIL:
(%•*type-class* ptr_1) = ptr_1 iff ptr_1 ϵ class whose name is *type-class*,
otherwise ().
See the section on the distinguished object **nil**, which follows for the NIL case.
Where *type-class* = { NUM | SMI | NIL | II | L | FP | CHAR | COMP |
A | RA | VEC | VP | VII | VF | STR | SC | SB | PR |
REC | ID | NORID | GENSYM | APPL | ABST | LAM |
MLAM | MU | ORCD | SD | FUN | FBPI | MBPI | SF |
UR | FUR | MUR | NT | PLEX }
Consult the type schema for a more complete hierarchical enumeration. These primitive type classes are disjoint and distinct.

The type constraints:

%•=*type-class* : *PTR* → *PTR* | ∇

(%•=*type-class* ptr_1) = ptr_1 iff ptr_1 ∊ class whose name is *type-class* otherwise ∇.

Notation: ∇ for run time detected domain error.

The allowable type-class names were mentioned in the type schema outline, and will be further elaborated in the descriptions of the the subclasses of the type hierarchy.

The type constraints differ from the type predicates in that the constraints are the guarantors of type. In the case that the value does conform it is passed through, otherwise an exception state is applied to the offending value. The nature of the exception handling is described in the interrupt section, returning with a proper value is just one of the actions possible. The important role that these constraint operators have is in defining functions whose parameters are constrained or whose value type is to be understood.

For example:

%(%•LAMBDA ((%•=SMI X)(Y . Z)) (%•=SMI *body*))

is the abstract description form for a function of precisely two arguments, the first of which must be a small integer, the second of which must be a pair, and the value of the function must be a small integer.

The constrained definitional process is not described in detail in this document. The user need not bother with constraints until such a time that their use and benefits are understood. At the time of this writing the full implementation of constraints is not yet in sight.

Access functions:

No access functions are provided to access the components of the pointer. To the extent that the pointer denotes an object of storage of some subtype, that storage object may have components which can be accessed by type specific access functions. The type specific access functions are described later.

Update functions:

Likewise, no functions are provided to update the fields of the pointer object. As stated with regard to access, the components of a denoted stored object may be updated by the type specific update functions.

Other primitives:

The output or canonical representation function:

%•PRINT : *PTR* × *STREAM* × *M* → *PTR* × *M*

(%•PRINT ptr_1 $stream_1$) $\{M_1\}$ = ptr_1 × $\{M_2\}$

where $ptr_1\{M_1\}$ = $\mathbf{s\text{-}exp}_1$

$stream_1$ $\{M_2\}$ is the result of

(%•WRITE $char_i$ $stream_1$) for each successive *char* in the canonical representation of $\mathbf{s\text{-}exp}_1$ as defined in the next section.

Prof. Dr. H. Stoyan
Universität Erlangen-Nürnberg
Institut für Mathematische Maschinen
und Datenverarbeitung (Informatik VIII)
Am Weichselgarten 9
91058 Erlangen

%•WRITE is a primitive stream operator and is defined in the section pertaining to streams.
Streams are an interpretation on pairs that serves to define, abstractly, a sequence of elements. Streams are primarily used for input and output.

The reasons for the primitive nature of %•PRINT are two fold: firstly it serves as a definition of the representations (***s-exp***) of **s-exp**: secondly it may be relied on by the system for system to user communications. In other words it defines a standard and ingrained notation. This has the drawback that for some programs it may be difficult to prevent that notation from showing through. This drawback seems common to computing systems with layered architecture. The benefits of layered architecture are substantial but no attempt to further justify this concept will be pursued here.

Syntactic representations for the data types

The traditional designation for the data objects of LISP is the term symbolic expressions or **s-exp**. The notation **s-exp** is used to denote the class of objects, and ***s-exp*** is used to denote the canonical representation form as a linear string of characters. For the most part **s-exp** bear a strong correspondence to the computational data types for **pointer** objects. The most obvious non-correspondence is that internal data types have a location handle which EQ is capable of comparing, but there is no comparable handle on **s-exps**. The transformations from data types to ***s-exp*** and vice versa do not in general preserve the EQ-equivalence class of the object. While the EQ-equivalence class is of considerable computational interest it is not state invariant.

The EQ-equivalence class of an object is equivalent to the place it is assigned when it is allocated. In an infinite memory this place could remain constant and would serve to simply denote each EQ-class object for all time in the history of a given memory. None the less, the place of an object (viewed as an object) has an interpretation which is meaningful only with respect to the memory in which it was created. Practically speaking, the memory is not infinite, and keeping track of an object's "creation number" would be prohibitively expensive. Indeed, it seems very difficult to conceive of a representation for the EQ-class objects which is memory invariant and in which commensurate objects are easily recognizable. One might also add that attendant to a memory place are other properties that the user would appreciate being abstracted from, such as: size, alignment, storage protection state, ...

LISP engenders a somewhat complex relationship between the internal computational domain and the abstract data object domain. It attempts to fool us into believing that we can operate in both isomorphically. It achieves necessary efficiencies by actually providing the computational EQ-class objects. The user (and also the system) then give these objects more abstract connotations as is the case with external representations.

S-exp do not reveal the EQ-class or place objects whereas, **pointers** do. The EQ-class objects are the most primitive, most computationally interesting, and most difficult to manage objects.

On the other hand, if the concern is for the structure of the object itself, and not the structure of the memory, we can consider the data objects as rooted directed graphs. Which was just what was done when we chose our external representation. This seems the most complete (yet memory independent) interpretation for data as objects. If we update a structure in memory it may no longer denote the same graph. But updating cannot in general extend beyond the memory, so it should not affect our choice of external representation. The sharing within the structure of an object is both representable and detectable. We therefore chose an external data model which shows both cyclic and acyclic sharing within the structure. Naturally, other models can be featured such as lists, and trees. While the latter will be more convenient from time to time, revealing the complete sharing within a structure is possible, shows more, is not memory dependent, and is in fact what is provided by default.

We are motivated in proposing the external notation to move the user from the pointer domain to an object domain. It is our goal to select an object domain with sufficient structure to be interesting, and efficient, yet as minimal as reasonable. Naturally there is in this selection a component of choice which does not rest entirely upon reason.

In the following description of the external syntax the hierarchical classification schema is slightly different than that used above in the internal data type schema. Here the *expression* language interpretation of the data is emphasized, rather than the relationship to underlying data primitives.

In the following syntax definitions, only the output or canonical form will be defined. This leaves some freedom to be permissive for the input forms. The definition of what is permissible input will not be given precisely at this time. Permissible input does naturally include canonical form.

{ and } are used for metalinguistic grouping.
{ and } are used for set braces.
[and] are used to indicate optionality.
| is used to separate alternatives.
Vertical alignment is also used for alternatives.
The ellipsis "..." is used to denote zero or more objects. Thus x... means zero or more x's, but ...x means zero or more of anything but x and then x.

'(' , ')' , '•' , '=' , '%' , ';' , ':' , '!' , '<' , '>' , '|' , '+' , '−' , and 'ƀ' are all used as special symbols in forming *s-expression* representations (also called *s-exp*). There are other isomorphic representations involving the choice of other characters. It is in the interest of communication that a single standard be chosen. The standard symbol goal is difficult to attain due to incompatibility of character sets, and the individual preferences among users.

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. $M \sim N$ for the complement of N in M; all points of M not in N.

Output Canonical Form

An *s-exp* is:

[*label*]{*c*ƀ | *id*ƀ | *funarg* | *combination* }
  where *label* is {*label-name* = }, and
    where *label-name* is {%L*digit*$_1$...*digit*$_n$} where $1 \leq n \leq 8$ and,
  where *id* ∊ *ID* the set of identifiers (names), and
  where *c* ∊ *C* the set of constants, and
  where *funarg* = %(%›FUNARG ƀ *e* ƀ • ƀ *sd*)
    where *sd* = (no syntactic form available or intended) ‡, and
  where *combination* = (*comp*$^{+}{}_{b}$ [ƀ •ƀ *comp*])
    where *comp* is {*label-name* | *c* | *id* | *funarg* | *combination* |
        {*label comp*} }

A *constant* is:
{*decimal-number* | *applicative-constant* | *nil* | *ranked-array* | *selector-structure*}
  where *decimal-number* = {*integer* | *floating-point* | *rational* | *complex* |
      interval }
    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 *rational* = %(/ ƀ *num* ƀ *denom*)†
      where *num* = *integer*
      where *denom* = *integer*
    where *complex* = %(i ƀ *r-part* ƀ *i-part*)†
      where *r-part* = *integer* | *floating-point*
      where *i-part* = *integer* | *floating-point*
    where *interval* = %(† ƀ *hi-end* ƀ *low-end*)†
      where *hi-end* = {*integer* | *floating-point* | *complex* | *rational*}

where *low-end* = {*integer* | *floating-point* | *complex* | *rational*}
where *applicative-constant* = { *abstraction* | *bpi* | *sf* | *ur* }
where *abstraction* = { *lambda-abstraction* | *mlambda-abstraction* |
operator-code-abstraction}
where *lambda-abstraction* =
%(%,LAMBDAƀ *bv* ƀ • ƀ *exp-seq*)
where *mlambda-abstraction* =
%(%,MLAMBDAƀ *bv* ƀ • ƀ *exp-seq*)
where *mlambda-abstraction* =
%(%,MLAMBDAƀ *bv* ƀ • ƀ *exp-seq*)
where *sequence-abstraction* =
%(%,SEQƀ *tag aux* ƀ • ƀ *ps-list*)
where *operator-code-abstraction* =
%(%,FR*CODEƀ *e* ƀ *f-list* ƀ • ƀ *lap-code*)
where *bpi* = †
where *sf* = %, { LAMBDA | MLAMBDA | MU | QUOTE | SETQ
| FUNCTION | LABEL | COND | SEQ | GO | EXIT
| PROGN | RETURN | FR*CODE | AUX | SETX }
where *ur* = {*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 *nil* = ()
where *ranked-array* = {*vector* | *string* }
where *vector* = {*pointer-vector* | *intermediate-integer-vector* |
floating-point-vector}
where *pointer-vector* = < $comp^{*}_{h}$ >
where *intermediate-integer-vector* = %I< $integer^{*}_{h}$ >
where *floating-point-vector* = %F < $floating\text{-}point^{*}_{h}$ >
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* = $digit^{+}$, and

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* = {*ntuple* | *plex*}
where *ntuple* = %(• ƀ *comp*$^*_ƀ$ [ƀ • ƀ *bit-string*]) †
where *plex* = %(•• ƀ *comp*$^*_ƀ$ [ƀ • ƀ *bit-string*]) †

An *id* is: {*norid* | *gensym* | *unid* | *mobid*}
where *norid* ∈ {x| {*non-num id-chr*...}}
where *non-num* ∈ {*ID-CHR* ~ *DIGIT*}
where *id-chr* = {*x*∈{*ANYCHR* ~ *IDDELIM*} | |*anychr*}
where *iddelim* = {ƀ | (|) | < | >}
where *gensym* = %G*gennum*
where *gennum* = *digit*$^+$
where *unid* = %*gennum*:*norid* †
where *mobid* = %•(*norid*ƀ • ƀ *directory*) †
where *directory* = †

The print representations will be described in more detail in the descriptions of the subclasses.

The input or ***s-exp*** recognition function:
%•READ : *STREAM* × *M* → *PTR* × *M*
(%•READ *stream*$_1$) *{M$_1$}* = *ptr*$_1$ × *{M$_2$}*
where *ptr*$_1$*{M$_2$}* = **s-exp**$_1$
stream$_1$ *{M$_2$}* is the result of
(%•NEXT *stream*$_1$) until sufficient *char* in the stream are recognized to form a ***s-exp*** representation, the corresponding value **s-exp**$_1$ is allocated.
%•NEXT is a primitive input output operator and is defined in the section pertaining to that topic.
%•READ is included as a basic primitive for recognizing ***s-exp*** and allocating **pointers**.

Input Syntax Commentary

There are many isomorphic forms for LISP input, for many applications unique parsers are required. The primitive understood operator for the LISP reader %•READ may provide additional conveniences, such as macro characters. While such extensions should not be curtailed, if allowed to supplant the standard external form, they will lead to the Tower of Babel phenomena. This applies to syntax sugaring extensions not to developments that correct some logical deficiency.

%•READ should be liberal about the use of blanks. Canonical form should enjoy a certain primitive status, %•READ should at the very least accept it.

With regard to the string delimiter character "¹" we have been accused of "adding to the Babeling" by not adopting """. ALGOL and certain versions of LISP have preferred "'". In defense of this convention it should be noted that "'" does enjoy a certain current popularity for this purpose. PL/1, FORTRAN, ASSEMBLER, APL, SNOBOL and IBM COBOL all use this convention. The underlying rational seems to be that in a minimal character set the "'" is a more useful character to have if a choice must be made.

A similar line of argument holds with regard to our choice of % as an extender of the character set. We have chosen this ugly minimal character set approach over the introduction of an ideal character set. This is unlike ALGOL 68 which "eschews" the problem.

These two operators are essential to the initial supervisor which is in essence: (%•SEQ () () TAG (%•PRINT (%•EVAL (%•READ USERINSTREAM)) USER-OUTSTREAM) (GO TAG))

The access equivalence relation: %•EQUAL : *PTR* x *PTR* → *TVAL*

(%•EQUAL ptr_1 ptr_2) = () iff

access$_1${ptr_1} = access$_1${ptr_2} and,

access$_1${ptr_2} = access$_1${ptr_1}

for each access$_1$ possible for both ptr_1 and ptr_2.
Informally, access functions traverse the underlying structure denoted by the pointer and retrieve a value, but do not update the memory. It should be noted that certain structures are presumed composite even if no access functions are provide to the user.

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.

But the two operand expressions are %•EQUAL and evaluated in the same context! Indeed the two expressions are themselves %•EQUAL but denote different values. This illustrates the difficulty that the EQ-class objects create.

If two pointer values are %•EQUAL then they both denote the same (possibly infinite) tree.

The update equivalence relation: %•EQUUP : *PTR* x *PTR* → *TVAL*

(%•EQUUP ptr_1 ptr_2) = () iff

(%•EQUAL ptr_1 ptr_2) ≠ () after any

update$_1${ptr_1}, and *update*$_1${ptr_2},

where *update*$_1$ is any allowed update operation on either ptr_1 or ptr_2.

If two pointer values are %•EQUUP they denote the same *expression*. That is, if they were each evaluated in the same context they would produce %•EQUUP values. Furthermore, if the independent %•PRINT representation of two pointers is the same then they are %•EQUUP and vice versa. Update equivalence is preserved by %•READ unless the structure contains a **gensym**.

If two pointers are %•EQUUP they both denote the same rooted, directed, graph.

We proceed with the elaboration of the descriptions of the types of the hierarchy of types:

SIMPLE OBJECTS

Simple-objects do not depend upon their components. That is, either they have no accessible components, or the properties of the class in question does not depend on the components.

There is some question as to whether or not **identifiers** should belong to this class. Because the truth values are included in this class it is not a distinct representation type class.

Abstract syntax: *SIMP{M}* = {**nil** U **decimal-number** U **character** U **truth-value** }

THE DISTINGUISHED NIL OBJECT

The **nil** object () is given the interpretation that it denotes the truth-value **false**. () is also commonly used as a list terminator and thought of as an empty list. The convention through out this system is to consider any non pair object as a suitable list terminator. () as a terminator does enjoy a certain exalted status in that (A • ()) prints as (A). Clearly, we consider () as denoting **false** and like any non-pair it is not a list. In the interest of the final elimination of reserved identifiers (for LISP370); the identifier NIL is not %•EQ to (). The variable NIL will be normally given the global value (). It may be necessary to convert all occurrences of NIL in old or foreign source files of ***s-exp's*** to (). Notice: There really is no empty list provided by the underlying implementation!

Pointer type class: *NIL* = (). A primitive type.

The type predicate: %•NIL

%•NIL : $PTR \rightarrow PTR$

(%•NIL ptr_1) = () iff $ptr_1 \neq ()$ otherwise ptr_2 where $ptr_2 \neq ()$ and denotes **true**.

There are no access, update or allocate functions for **nil**. In certain other LISP systems the access functions CAR and CDR are well defined if applied to **nil** but always yield the value **nil**. But in these very same systems the pair updating operators are not well defined when applied to **nil**. This inconsistent approach would seem to complicate the scheme for constrained types. The point of view of this system is that **nil** is a unique non-pair used to denote falsity whose representation makes its use as list terminator result in a simple list.

NUMBERS

Pointer type class: *NUM*

Abstract syntax: *NUM{M}* = {**integer** ∪ **floating-point** ∪ **rational**† ∪ **complex**† ∪ **interval**†}

LISP/370 currently operates on three basic types of numbers, and several other types of numbers are anticipated. A basic numeric data item may be an integer or a floating-point. Integers are stored in one of two possible formats, and each is denoted by a pointer of a distinct type, depending upon their value. In the range -2^{27} to $2^{27}-1$ ($-134,217,728$ to $134,217,727$), the **small-integer** type is used (see Small Integer Format). This type pointer (designated *SMI*) encodes the numeric value as immediate data, and so achieves greater efficiency in computation and storage than the **large-integer** format which is used for all other integer values. All integers are stored exactly by LISP. The only limitation on size is the available space in the heap.

Primitives:

Type predicate: %•NUM

The type constraint: %•=NUM

(%•=NUM ptr_1) = ptr_1 iff $ptr_1 \in NUM$, otherwise ∇.

Generic Arithmetic Operators:

operator	operation	type of operands	type of result
%:+	addition (%:+ *m* ...) = Σ*m*...	*floating-point* or *integer*	*integer*, if all operands are of type *integer*, *floating-point* otherwise
%•−	subtraction (%•− *m n*) = *m*−*n*	*floating-point* or *integer*	*integer*, if all operands are of type *integer*, *floating-point* otherwise
%:*	multiplication (%:* *m*...) = Π*m*...	*floating-point* or *integer*	*integer*, if all operands are of type *integer*, *floating-point* otherwise

%•/	quotient (%•/ *m n*) = *m*÷*n*	*floating-point* or *integer*	*integer*, if all operands are of type *integer*, *floating-point* otherwise
%•DIV	decimal division (%•DIV *m n*) = *m*÷*n*	*floating-point* or *integer*	*floating-point*
%•**	m^n	*floating-point* or *integer*	*floating-point integer*, if all operands are of type *integer*, *floating-point* otherwise
%•MOD	remainder mod *n* (%•MOD *m n*) = $m - n*(m \div n)$	*integer*	*integer*
%•<	less than predicate (%•< *m n*) = $m < n$	*floating-point* or *integer*	*TVAL* *m* if true, otherwise ().
%•>	greater than predicate (%•> *m n*) = $m > n$	*floating-point* or *integer*	*TVAL* *m* if true, otherwise ().
%•>=	not less than predicate (%•>= *m n*) = $m \geq n$	*floating-point* or *integer*	*TVAL* *m* if true, otherwise ().
%•<=	not greater than predicate (%•<= *m n*) = $m \leq n$	*floating-point* or *integer*	*TVAL* *m* if true, otherwise ().
%•=	equality predicate (%•= *m n*) = $m=n$	*s-exp*	*TVAL*
%•CHS	change sign (%•CHS *m*)	*floating-point* or *integer*	*integer*, if operand is of type *integer*, *floating-point* otherwise
%•ABS	absolute value (%•ABS *m*) = $\|m\|$	*floating-point* or *integer*	*integer*, if operand is of type *integer*, *floating-point* otherwise
%•=0	zero predicate (%•=0 *m*)	*floating-point* or *integer*	0 if operand is 0, otherwise ()

%•<0	negative number predicate (%•<0 *m*)	*floating-point* or *integer*	operand if operand is less than 0, otherwise ()

Small Integers:

Small Integer Pointer Format:

0011	S - - -	- -

S is a sign bit (1 for negative value, in two's complement form);
- represents a data bit which is part of the actual numeric value.

Note that a small integer is actually a (non-stored) pointer value. It is not a reference to another data object.

Pointer type class: *SMI*

Abstract syntax: *SMI{M}* = **smi** = $-134{,}217{,}728 \leq$ **integer** $\leq 134{,}217{,}727$. A distinguished primitive class.

Primitives:

Type predicate: %•SMI

The type constraint: %•=SMI

(%•=SMI ptr_1) = ptr_1 iff $ptr_1 \in SMI$, otherwise ∇.

Small integers are allowed inputs to and outputs of the ordinary generic arithmetic functions. In this role the **smi** are a subrange of the **integers**.

Functions over the commutative ring s (an isomorph of **smi**):

%:S+ , %•S– , %:S* , %•S/ , %•SMOD , %•S** , %•SCHS , %•SABS , %•S<0 , %•S> , %•S< , %•S<= , %•S>= , %•S=0 , and %S.= .

These operators do arithmetic modulo 2^{28} but the two's complement notation results in numbers greater than $(2^{27}-1)$ being considered negative.

We denote this domain s = $\{-2^{27}, -2^{27}+1, \ldots, 0, 1, \ldots, 2^{27}-1\}$.

For $\oplus_s \in$ { %:S+ , %•S−, %:S* , %•S/ , %•SMOD , %•S** }, and
$\oplus \in$ { %:+ , %•−, %:* , %•/ , %•MOD , %•** };

$s_1 \oplus_s s_2 = -2^{27}$ if $s_1 \oplus s_2 = -2^{27}$, and
(%•MOD $\{s_1 \oplus s_2\}$ 2^{27}) otherwise.

For $\oplus_s \in$ { %•SCHS , %•SABS }, and
$\oplus \in$ { %•CHS , %•ABS };

$\oplus_s s_1 = -2^{27}$ if $\oplus s_1 = -2^{27}$, and
(%•MOD $\{\oplus s_1\}$ 2^{27}) otherwise.

Large Integers:

Large Integer Format:

LCBVTP		Vector Length in Bytes
0	Low-order Digit (radix 2^{31})	
	⋮	
0	High-order digit (radix 2^{31})	

The format pictured above defines the magnitude of a large integer. There are two pointer type codes which designate large integers; one indicates a positive large integer, the other indicates a negative large integer. Because these type codes are not in the class of vectors, it is not possible to select an element (digit) of a large integer with vector functions such as %•ELT.

Pointer type class: *L* a distinguished primitive class.

Abstract syntax: ***L{M}*** **= lint** = **{integer** < −134,217,728} U **{integer** > 134,217,727}.

Primitives:

Type predicate: %•L

Type predicate: %•L

The type constraint: %•=L

(%•=L ptr_1) = ptr_1 iff $ptr_1 \in L$, otherwise ∇.

Large integers are allowed inputs to and outputs of the ordinary generic arithmetic functions. There is not provided any type specific basic constant functions with either domain or range constrained to the type *L*.

Integers

Pointer type class: *I*

Abstract syntax: *I{M}* = **integer** = **{smi ∪ lint}**

Type predicate: %•I

The type constraint: %•=I

(%•=I ptr_1) = ptr_1 iff $ptr_1 \in I$, otherwise ∇.

Primitives:

%•ODDP: $I \rightarrow TVAL$
(%•ODDP *i*) = () if ((*i*MOD2)=0) otherwise *i*.

The following generic operators have integer values when given integer arguments:

%:+ , %•− , %:* , %•/ , %•MOD , %•CHS , %•** , %•ABS

Floating Point Numbers

The user can (at some peril) view the floating point numbers as real numbers whose decimal representation was truncated some number of places to the right of the decimal point. Until the computer provides efficient decimal floating point hardware that has no perils, we will be content to use the available hexadecimal floating hardware. This decision will affect (in a hopefully minor way) our ideals for conversion to canonical output form and our understanding of the rules of arithmetic. Floating-point numbers are stored using System/370 double precision floating point format, yielding 53 to 56 bits of precision for the mantissa and a range of up to (about) 10^{74}. Floating-point numbers are stored in a separate section of the heap used only for these data. This area is allocated at the high address end of the space reserved for the heap, and extends toward lower addresses as new floating-point numbers are generated.

Pointer type class: *FP* a distinguished primitive class.

Abstract syntax: *FP{M}* = **float** shorthand for **truncated-real**

Type predicate: %•FP

The type constraint: %••=FP

(%••=FP ptr_1) = ptr_1 iff $ptr_1 \in FP$, otherwise ∇.

When provided with floating arguments the following generic operators have floating values:

%:+ , %•– , %:* , %•QUOT , %•/ , %•CHS , %•** , %•ABS

Primitive operators whose domain and range is restricted to the floats:

%:FP+ , %•FP– , %:FP* , %•FP/ , %•FP** , %•FPCHS , %•FPABS
%•FP<0 , %•FP> , %•FP< , %•FP<= , %•FP>= , and %•FP=0

The print representation for a floating-point number always includes a decimal point to distinguish floating-points from integer values. This decimal point must be preceded by at least one decimal digit, to avoid possible confusion with the period used in printing pairs. A minus sign may precede the first digit to indicate a negative value.

Both integer and floating-point numbers may be followed by a decimal exponent formed by the letter E, a plus or minus sign (plus is optional), and the exponent magnitude expressed in decimal digits.

There are two parameters, **FUZZ** and **NDIGITS**, which control the way in which floating-point numbers are translated into their print representations for output. **FUZZ** refers to a value used to define the intended precision of floating-point number operations. Two numbers, X and Y, are equal in the LISP system if

|X - Y| <= **FUZZ** * maximum (|X| , |Y| , 1.0)

Insofar as printing a floating-point number, X, is concerned, a character representation is generated for the value in the range

X-FUZZ*|X| to X+FUZZ*|X|

which results in the shortest character string. This print representation may include an exponent, in which case there will be exactly one decimal digit before the decimal point, or in cases where the number of digits (exclusive of decimal point and a possible minus sign) needed to represent the numeric value is less than **NDIGITS**, no exponent will be printed and the decimal point will be placed wherever is required.

The user may specify values for **FUZZ** and **NDIGITS** by using the function %•SETFUZZ.

%•SETFUZZ : $PR \times PR \rightarrow PR \times M$

(%•SETFUZZ pr_1)$\{M_1\}$ = $pr_2\{M_2\}$

where $pr_1\{M_1\} = (float_1 \bullet smi_1)$ and, $pr_2\{M_2\} = (float_0 \bullet smi_0)$.

FUZZ$\{M_1\} = float_0$,

FUZZ$\{M_2\} = float_1$,

NDIGITS$\{M_1\} = smi_0$,

NDIGITS$\{M_2\} = smi_1$,

CHARACTER OBJECTS

The **characters** are an understood subrange of the **identifiers**. They are not a primitive type.

Pointer type class: *CHAR*

Abstract syntax: $CHAR\{M\}$ = {**character**}

$CHAR \subset ID$

Primitives:

Type predicate: %•CHAR

The type constraint: %•=CHAR

(%•=CHAR ptr_1) = ptr_1 iff $ptr_1 \in CHAR$, otherwise ∇.

Character object to EBCDIC character code: %•CHIDEBCD

%•CHIDEBCD : $CHAR \rightarrow SMI$

(%•CHIDEBCD $char_1$) = smi_1

where $0 \leq smi_1 \leq 255$ and the correspondence is defined in IBM form number GX20-1850-2 System/370 Reference Summary (the yellow card).

EBCDIC to character object: %•EBCDCHID

%•EBCDCHID : $SMI \rightarrow CHAR$

(%•EBCDCHID smi_1) = $char_1$

where $0 \leq smi_1 \leq 255$ and the correspondence is defined in IBM form number GX20-1850-2 System/370 Reference Summary (the yellow card).

TRUTH-VALUES

Truth-values are an interpretation on the set of all data objects. They are not a distinct primitive class. If the object is () then we interpret it as **false**, otherwise it denotes **true.**

Pointer type class: $TVAL = PTR$

Abstract syntax: $TVAL\{M\}$ = {**true** ∪ **false**}

Primitives:

Truth value: %•TVAL

%•TVAL : $PTR \rightarrow \{ 1 \mid 0 \}$

(%•TVAL ptr_1) = 0 iff ptr_1 = (), otherwise 1.

Comment: The truth values are an interpretation on the total *PTR* domain. The justification for this is that in the language syntax and semantics the predicate expressions of the conditional are first class i.e. any expression can be written in that place. This results in an elegance and versatility that can be appreciated from the point of view of theory and pragmatics.

COMPOSITE OBJECTS

The composite objects are viewed as having components even if no access function is provided. In the case of objects where no access function is provided the access-equivalence operator %•= may consider otherwise inaccessible components as comparable.

Pointer type code: $COMP = \{ARRAY \cup RECORD \cup COMPLEX\}$

Abstract syntax: $COMP\{M\}$ = {**array** ∪ **record** ∪ **complex** }

ARRAYS

Arrays are objects that have components of all the same type.

Pointer type class: $A = \{PR \cup RA\}$

Abstract syntax: $A\{M\}$ = { **pair** ∪ **ranked-array**}

Primitives:

Type predicate: %•A

The type constraint: %•=A

(%•=A ptr_1) = ptr_1 iff ptr_1 = ϵ *A*, otherwise ∇.

PAIRS

LISP/370 implementation format of the storage object:

XXXX	YYYY	AAAAAAAAAAAAAAAAAAAAAAAA
XXXX	YYYY	AAAAAAAAAAAAAAAAAAAAAAAA

A **pair** is a stored data object having two component pointer objects which are referred to as the CAR component and the CDR component (for historical and compatibility reasons). The storage allocation for a pair is two contiguous full-words. Both of these words contain pointers. The CAR component occupies the first word; the CDR component occupies the second word. Since a pointer is used to represent any LISP data object, a pair is an association of two completely arbitrary LISP data objects.

Pointer type class: *PR*. A distinguished primitive class.

Abstract syntax: ***PR{M}* = pair = s–exp × s–exp**

Primitives:

Type predicate: %•PR

The type constraint: %•=PR

(%•=PR ptr_1) = ptr_1 iff ptr_1 = ϵ *PR*, otherwise ∇.

Access functions: %•CAR and %•CDR. Two basic functions are provided for selecting part of a pair. %•CAR and %•CDR applied to a pair return as their value the corresponding component of the pair.

%•CAR : *PR* → *PTR*

(%•CAR pr_1) = ptr_2

where $pr_1 = (ptr_2 \bullet ptr_3)$

This is another example of a domain restricted primitive. It is defined only over the pair domain. The definition of these primitives when the domain is not conformal is ∇. As the notation is meant to imply this it is not explicitly stated.

%•CDR : *PR* → *PTR*

(%•CDR pr_1) = ptr_3
    where $pr_1 = (ptr_2 \bullet ptr_3)$

Allocation function:

%•CONS : $PTR \times PTR \times M \rightarrow PR \times M$
    (%•CONS ptr_1 ptr_2) $\{M_1\} = pr_3$ $\{M_2\} = (ptr_1 \bullet ptr_2)$
        where $pr_3 \notin PTR$ of M_1, and $pr_3 \in PTR$ of M_2, or simply, a new pointer is allocated.
    (%•CAR pr_3) $\{M_2\} = ptr_1$
    (%•CDR pr_3) $\{M_2\} = ptr_2$

Update functions: %•RPLACA and %•RPLACD.

%•RPLACA : $PR \times PTR \times M \rightarrow PR \times M$
    (%•RPLACA pr_1 ptr_2) $\{M_1\} = pr_1$ $\{M_2\}$
    pr_1 $\{M_1\}$ = (x • y)
    (%•CAR pr_1) $\{M_2\} = ptr_2$
    pr_3 $\{M_2\} = pr_3\{M_1\}$
        for all $pr_3 \in PTR$ of M_1, such that pr_1 ***indep*** pr_3 .
            Where the independence relation ***indep*** expresses the notion of having no shared component.
            x ***indep*** x is false
            if x ***indep*** x', then x' ***indep*** x
            if x ***indep*** x' then, x ***indep*** x'_i,
                for all components x'_i of x'.
    Simply stated, M_1 and M_2 differ only in the meaning of the composite objects that share pr_1 as a component and not necessarily in those.

%•RPLACD : $PR \times PTR \times M \rightarrow PR \times M$
    (%•RPLACD pr_1 ptr_2) $\{M_1\} = pr_1$ $\{M_2\}$
    pr_1 $\{M_1\}$ = (x • y)
    (%•CDR pr_1) $\{M_2\} = ptr_2$
    pr_3 $\{M_2\} = pr_3\{M_1\}$
        for all $pr_3 \in PTR$ of M_1, such that pr_1 ***indep*** pr_3 .

The primitive print representation of a pair is a left parenthesis followed by the print representation of the first element of the pair, a blank, a period, a blank, the print representation of the second element of the pair, and finally a right parenthesis.

(comp ƀ • ƀ comp)

In most cases, however, a more complex print representation is used. These abrogations of the above rule occur for economy of representation and because of the desire to emphasize the list interpretation of these pair structures.

RANKED ARRAYS

Pointer type class: $RA = \{R1A \ \cup \ MATRIX\}$

Abstract syntax: $RA\{M\}$ = {**rank-one-array** ∪ **matrix**}

Primitives:

Rank function: %•RANK

%•RANK : $PTR \rightarrow SMI$

(%•RANK ptr_1) =

0 iff $ptr_1 \notin RA$, and

1 iff $ptr_1 \in R1A$, and

n iff $ptr_1 \in MATRIX$,

where n is the number of dimensions.

ARRAYS OF RANK ONE

LISP/370 **rank-one-arrays** are composite stored objects that have { 0, ..., n−1} as an index set, also they have components of uniform type. Like pointers, this class is not a primitive storage type but rather a union of subtypes.

Pointer type class: $R1A = \{LIST \cup STR \cup VEC\} = PTR$

Abstract syntax:

$R1A\{M\}$ = **rank-one-array** = { **list** ∪ **string** ∪ **vector**} = **comp**^{n}

where **comp** the components are **s-exp**

and **comp**^{0} implies an empty rank one array has an empty index set.

Because we allow the conventional interpretation that any non-pair has an interpretation as an empty list; **rank-one-arrays** = **s-exp**.

Notation: $\boldsymbol{a}^n$ for $\boldsymbol{a} \times \boldsymbol{a} \times \ldots$ (n factors).

Rank-one-arrays are classified as **vectors**, **strings**, and **lists**. **Vectors** are characterized their cardinality, the number of elements they contain. The cardinality may be computed by the operator %•NC. **Strings** on the other hand are characterized by current cardinality and capacity for extension. The capacity for extension of a string may be computed by %•CAP . **Lists** are, as previously explained, an interpretation on **s-exp**. They are dynamically extendible to the limit of available space. These objects will be described in detail below.

Except for bit strings, rank one arrays may have any length for which sufficient space exists in the heap. Bit vectors (in LISP/370) may have a maximum of 2^{24}-1 (16,777,215) elements (bits).

Primitives:

Type predicate: Not provided because this is not a distinct type.

Type constraint: Not provided because this is not a distinct type.

Cardinality function:

%•NC : $RIA \rightarrow SMI$
(%•NC rla_1) = smi_1
$smi_1\{M\}$ = n , where $rla_1\{M\}$ = $\mathbf{ptr}^n$.

Access function:

%•ELT : $RIA \times SMI \rightarrow PTR$
(%•ELT rla_1 smi_1) = iff $0 \leq i < n-1$ then ptr_i otherwise ∇,
where $rla_1\{M\}$ = $\mathbf{ptr}^n$ = $\mathbf{ptr}_0, \ldots, \mathbf{ptr}_{n-1}$, and $smi_1\{M\}$ = i.

Notation: ∇ for run time detected domain error.

All rank one arrays use zero-origin indexing for identification of their components. The function %•ELT is a general rank one array accessing function, applicable to any type of rank one array. Provided, of course, the index is within bounds. Thus

(%•ELT rla 0)

is always the first element of rank one array. Other accessing functions, tailored to a particular type of rank one array, are provided because they are more efficient in execution. These are each described in the section about that rank one array.

Update function:

%•SETELT : $RIA \times SMI \times PTR \times M \rightarrow PTR \times M$
(%•SETELT rla_1 smi_1 ptr_1)$\{M_1\}$ = iff $0 < i < n$ then $ptr_1 \times M_2$ otherwise ∇,
where $rla_1\{M\}$ = $\mathbf{ptr}^n$, and $smi_1\{M\}$ = i.
(%•ELT rla_1 smi_1)$\{M_2\}$ = ptr_1 , and
(%•ELT rla_1 smi_2)$\{M_2\}$ = (%•ELT rla_1 smi_2)$\{M_1\}$
for all smi_2 = 0,..., n−1 where $smi_2 \neq smi_1$, and
$ptr_2\ \{M_2\}$ = $ptr_2\{M_1\}$
for all $ptr_2 \in PTR$ of M_1, such that ptr_1 ***indep*** ptr_2 .

LISP/370 structures may also be classified as: Pointer component vectors and no-pointer component vectors. Pointer component vectors, as the name implies, may contain references to any LISP data objects (including themselves, so circular structures are possible). Pointer component vectors are **pointer-vectors** and **lists**.

No-pointer vectors contain only binary information -- that is, data which cannot contain references to other data objects. Thus, no-pointer vectors are non-descendable from the point of view of the garbage collector and structure-dependent functions such as %•EQUAL and %•PRINT. No-pointer vectors are: **bit-strings**, **character-strings**, **intermediate-integer-vectors**, and **floating-point-vectors**.

Pairs As Lists

The abstract data structure **list** is usually defined as:
(**list-element**...) where () the **empty-list** is a list.
This system does not have a distinct *LIST* type class, therefore the domains and ranges of functions, and the domains of variables cannot be constrained by primitive constraint functions to this type class. On the other hand there are a great number of functions pertaining to lists.

Lists are composite objects created by applying a conventional interpretation to **s-exp**. Thus each pair is a list and any non-pair is an empty list. The CAR component of the pair is interpreted as a **list-element** of that list, and the CDR component of the pair is interpreted as the remainder of that list. The distinguished nil object () is commonly used to denote the empty list. Thus, if the CDR of a pair is not a pair, there are no remaining elements in that list. Note: The CDR could be some other object, not a pair and not **nil**. These empty lists are also **lists**! Because *LIST* is isomorphic to *PTR*, no *type-class* predicate is provided nor is the type-constraint.

There are few strict **list** domain or range functions; the list functions provided by the system are total functions. In this mode of interpretation all non-pair objects (*{PTR ~ PR}*) may be used to terminate a list.

It may surprise the reader that LISP does not have **lists** as a distinguished type. **Lists** are abstract data structures that originally were thought of as the principle data structure for the language. When pragmatic concerns about insertion were considered the pointer-pair schemes became an attractive solution. In most LISP systems today what we have is a *list notation* for **pairs**.

For the purposes of functions which have a list interpretation on pairs, the CDR component of the last pair of the list is not considered to be part of the list. Because of this somewhat liberal interpretation of what is a list, a recursive function over pair structures that only uses the "()" test as a termination test is only well defined for "()" terminated lists. The system utility functions all use the "not pair" termination test for partial functions on lists.

The print representation of a list is a modification of the representation of its component pairs as described above. This modification is intended to improve readability by eliminating some of the parentheses while still divulging the sharing of data; however, the inclusion of some (or all) of the deleted parentheses is always acceptable in input data. This list notation form can be most simply understood as an elision rule applied to the pure dotted pair notation:

> whenever " • (" occurs it may be replace by a blank and the balancing ")" deleted.

This seems more complicated when described in words than when illustrated by example.

Thus, the list

```
(A • (B • (C • ())))
```

would appear as

```
(A B C)
```

when printed.

Since a pair is a perfectly reasonable element of a list, it is possible to create lists which include themselves, or parts of themselves, as elements. LISP/370 uses a general scheme for input/output which indicates the sharing of data. This sharing scheme, as well as other aspects of the LISP/370 input/output system, makes use of a *break character* which is defined in the standard system as percent (%). An input expression written:

```
%L1=(A • %L1)
```

generates a pair whose CAR component is a pointer to the identifier A and whose CDR component is a pointer to the pair itself. The list interpretation of this pair would be a circular list, effectively an infinite list of A's. A function meant to traverse lists might be non-terminating for this object.

This sharing notation need not generate a circular list. For example, the expression:

```
(%L1=(A) %L1)
```

generates a list containing two elements. The first element is the list containing a single element -- the identifier A -- and the second element is another identical pointer. This is to be distinguished from the expression:

```
((A) (A))
```

which also generates a list of two elements, each of which is a list containing the single identifier A. In this case, however, the two elements are different pointers, although they point to equal (but separately stored) lists.

For purposes of accessing the elements of the list, both expressions are equivalent (but note that the list having the shared data requires less storage). These two lists are not equivalent with respect to updating. That is, the product of updating one may not be the same as the product achieved by the same updating operation applied to the other.

VECTORS

Pointer type class: VEC = $\{VP \cup VII \cup VF\}$

Abstract syntax:
VEC{M} = **vector** = { **pointer-vector** ∪ **intermediate-integer-vector** ∪ **floating-point-vector**}

Type predicate: %•VEC

Type constraint: %•=VEC
(%•=VEC ptr_1) = ptr_1 iff $ptr_1 \in FP$ otherwise ∇.

Reference vectors:

Reference Vector Format:

LCRVTP	Vector Length in Bytes
Pointer for component 0	
Pointer for component 1	
⋮	
Pointer for Last component	

Pointer type class: *VP*. A distinguished primitive class.

Abstract syntax: *VP{M}* = **pointer-vector** = $\textbf{pointer}^n$

Type predicate: %•VP

Type constraint: %•=VP
(%•=VP ptr_1) = ptr_1 iff $ptr_1 \in VP$ otherwise ∇.

Access function:

%•VPELT : $VP \times SMI \rightarrow PTR$

(%•VPELT vp_1 smi_1) = iff $0 \leq i < n-1$ then ptr_i otherwise ∇,

where $vp_1\{M\} = \mathbf{ptr}^n = \mathbf{ptr}_0, \ldots, \mathbf{ptr}_{n-1}$, and $smi_1\{M\} = i$.

Update function:

%•VPSET : $VP \times SMI \times PTR \times M \rightarrow PTR \times M$

(%•VPSET vp_1 smi_1 ptr_1)$\{M_1\}$ = iff $0 \leq i < n-1$ then $ptr_1 \times M_2$ otherwise ∇,

where $vp_1\{M\} = \mathbf{ptr}^n = ptr_0, \ldots, ptr_{n-1}$, and $smi_1\{M\} = i$.

(%•VPELT vp_1 smi_1)$\{M_2\} = ptr_1$, and

(%•VPELT vp_1 smi_2)$\{M_2\}$ = (%•VPELT vp_1 smi_2)$\{M_1\}$

for all $smi_2 = 0, \ldots, n-1$ where $smi_2 \neq smi_1$, and

$ptr_2\{M_2\} = ptr_2\{M_1\}$

for all $ptr_2 \in PTR$ of M_1, such that ptr_1 ***indep*** ptr_2 .

The pointer vector allocator:

%•VPGET : $SMI \times M \rightarrow VP \times M$

$smi_l\{M\} = k$

(%•VPGET smi_l) $\{M_l\} = vp_l\{M_2\} = <()_0 \cdots ()_{k-1}>$

$M_2 \sim M_l = vp_l\{M_2\}$

$vp_l \notin PTR$ of M_l

Allocates an pointer vector with smi_l elements all ().

The print format of a pointer vector uses angle brackets to delimit the extent of the vector and blanks to separate components of the vector:

<***comp***$_0$ ***comp***$_1$ ... ***comp***$_n$> and < > denotes the empty pointer vector,

where ***comp***$_i$ is the print representation of the LISP data object referenced by the i'th component of the reference vector.

Intermediate-integer-vectors:

Intermediate-integer-vectors Format:

LCRVTP	Vector Length in Bytes
Intermediate-integer for component 0	
Intermediate-integer for component 1	
. . .	
Intermediate-integer for Last component	

Pointer type class: *VII*

Abstract syntax: *VII{M}* = **intermediate-integer-vector = intermediate-integer^n**

Type predicate: %•VII

Type constraint: %•=VII

(%•=VII ptr_1) = ptr_1 iff $ptr_1 \in VII$ otherwise ∇.

Access function:

%•VIIELT : $VP \times SMI \rightarrow II$

(%•VIIELT vii_1 smi_1) = iff $0 \leq i < n-1$ then ii_i otherwise ∇,

where $vii_1\{M\}$ = ii^n = $ii_0, \ldots, ii_{n-1}$, and $smi_1\{M\}$ = i.

Update function:

%•VIISET : $VII \times SMI \times II \times M \rightarrow II \times M$

(%•VIISET vp_1 smi_1 ii_1)$\{M_1\}$ = iff $0 \leq i < n-1$ then $ii_1 \times M_2$ otherwise ∇,

where $vii_1\{M\}$ = ii^n = $ii_0, \ldots, ii_{n-1}$, and $smi_1\{M\}$ = i.

(%•VIIELT vii_1 smi_1)$\{M_2\}$ = ii_1 , and

(%•VIIELT vii_1 smi_2)$\{M_2\}$ = (%•VIIELT vii_1 smi_2)$\{M_1\}$

for all smi_2 = 0,..., n−1 where $smi_2 \neq smi_1$, and

ptr_2 $\{M_2\}$ = $ptr_2\{M_1\}$

for all $ptr_2 \in PTR$ of M_1, such that ptr_1 ***indep*** ptr_2 .

The intermediate-integer-vector allocator:

%•VIIGET : $SMI \times M \rightarrow VII \times M$

$smi_l\{M\}$ = k

(%•VIIGET smi_l) $\{M_l\}$ = $vii_l\{M_2\}$ = %I<$0_0 \ldots 0_{k-1}$>

$M_2 \sim M_l$ = $vii_l\{M_2\}$

$vii_l \notin PTR$ of M_l

Allocates an intermediate-integer vector with smi_l elements all zero.

The print format of an intermediate integer vector uses angle brackets to delimit the extent of the vector and blanks to separate components of the vector:

%I<ii_0 ii_1 ... ii_n> and %I< > denotes the empty intermediate integer vector.

where ii_i is the print representation of the LISP intermediate integer referenced by the i'th component of the vector.

Floating-point Vectors:

Floating-point Vector Format:

LCRVTP	Vector Length in Bytes
Floating-point for component 0	
Floating-point for component 1	
. . .	
Floating-point for Last component	

Pointer type class: *VF*. A distinguished primitive class.

Abstract syntax: $VF\{M\}$ = **floating-point-vector** = **floating-point**^{n}

Type predicate: %•VF

Type constraint: %•=VF

(%•=VF ptr_1) = ptr_1 iff $ptr_1 \in VF$ otherwise ∇.

Access function:

%•VFELT : $VF \times SMI \rightarrow FP$

(%•VFELT vf_1 smi_1) = iff $0 \leq i < n-1$ then fp_i otherwise ∇,

where $vf_1\{M\}$ = $\mathbf{vf}^n = \mathbf{fp}_0, \ldots, \mathbf{fp}_{n-1}$, and $smi_1\{M\} = i$.

Update function:

%•VFSET : $VF \times SMI \times FP \times M \rightarrow FP \times M$

(%•VFSET vf_1 smi_1 fp_1)$\{M_1\}$ = iff $0 \leq i < n-1$ then $fp_1 \times M_2$ otherwise ∇,

where $vf_1\{M\}$ = $\mathbf{fp}^n = fp_0, \ldots, fp_{n-1}$, and $smi_1\{M\} = i$.

(%•VFELT vf_1 smi_1)$\{M_2\}$ = vf_1 , and

(%•VFELT vf_1 smi_2)$\{M_2\}$ = (%•VFELT vf_1 smi_2)$\{M_1\}$

for all smi_2 = 0,..., n−1 where $smi_2 \neq smi_1$, and

$ptr_2\ \{M_2\} = ptr_2\{M_1\}$

for all $ptr_2 \in PTR$ of M_1, such that ptr_1 ***indep*** ptr_2 .

The floating-point vector allocator:

%•VFGET : $SMI \times M \rightarrow VF \times M$

$smi_1\{M\} = \mathrm{k}$
(%•VFGET smi_1) $\{M_1\} = vf_1\{M_2\}$ = %F<0.$_0$... 0.$_{k-1}$>
$M_2 \sim M_1 = vf_1\{M_2\}$
$vf_1 \notin PTR$ of M_1

Allocates a floating-point vector with smi_1 elements all zero.

The print format of an floating-point vector uses angle brackets to delimit the extent of the vector and blanks to separate components of the vector:

%F<**fp**$_0$ **fp**$_1$ ... **fp**$_n$> and %F< > denotes the empty floating-point vector,

where **fp**$_i$ is the print representation of the LISP floating point number denoted by the i'th component of the vector.

STRINGS

Strings (character and bit vectors) share a special storage characteristic in the LISP/370 system. For reasons of economy (of both storage and processing time) they are stored in contiguous blocks of storage. Nevertheless, because it is considered desirable to allow them to vary in length, a compromise has been achieved which involves maintaining two separate pieces of length information for each string. One length reflects the amount of storage allocated for the string, in terms of the number of elements which may be put into the string without having to allocate more storage for a larger string. The other length refers to the current number of elements which are actually present, the cardinality, which is less than or equal to the capacity of the string.

Pointer type class: *STR*

Abstract syntax: *STR{M}* = {**character-string** ∪ **bit-string**}

Primitives:

The capacity function: %•CAP

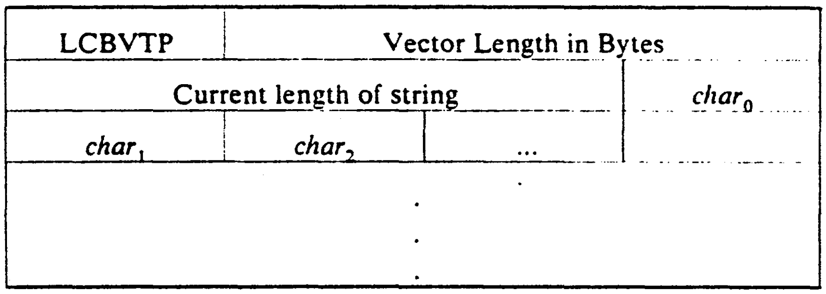
%•CAP : $STR \rightarrow SMI$
(%•CAP str_1) = smi_1 = k, where
$str_1\{M\}$ =
Case 1: %k'$char_0$...$char_{n-1}$'
Case 2: '$char_0$...$char_{k-1}$'
Case 3: %Bk[:m]'hex_0...hex_{n-1}'
Case 4: %B'hex_0...hex_{n-1}', and k=4n.

The change cardinality (NC) function: %•CHGNC

%•CHGNC : $STR \times SMI \times M \rightarrow \{STR \times M \mid \nabla\}$

(%•CHGNC str_1 smi_1)$\{M_1\}$
  = ∇ iff (%•CAP str_1) $\geq$ {smi_1+(%•NC str_1)$\{M_1\}$} ≥ 0,
  otherwise $str_1\{M_2\}$,
    where (%•NC str_1)$\{M_2\}$ = {smi_1+(%•NC str_1)$\{M_1\}$}

Character Vector Format:



Pointer type class: *SC*. A distinguished primitive class.

Abstract syntax: $SC\{M\}$ = {**character**^{n}}

Primitives:

Type predicate: %•SC

The type constraint: %•=SC

(%•=SC ptr_1) = ptr_1 iff ptr_1 = ϵ *SC*, otherwise ∇.

The character string allocator:

%•SCGET : $SMI \times M \rightarrow SC \times M$
  (%•SCGET smi_l) $\{M_l\}$ = $sc_l\{M_2\}$ = %k''
  k = (((%•MOD (smi_l+6) 4) × 4) − 3)
  $M_2 \sim M_l = sc_l\{M_2\}$
  $sc_l \notin PTR$ of M_l
Allocates a character string with capacity for at least smi_l characters.

There are two input/output representations for character vectors. The more general format is:

%k'c...c' .

where 'k' is the maximum number of characters which could be put into the vector for the character string being read or printed. The actual contents of the character string 'c...c' reflects only the current length of the string, and might be null. Any character may be included as part of a character string; however, the string delimiter character and the letterizer character must be treated specially. In order to avoid confusion about whether a string delimiter character actually delimits a string or is intended as a data character in a string, every occurrence of the string delimiter character as a data character in a string must be prefixed by a letterizer character. This letterizer character is not part of the character string in storage; it is created during output by the print routine, and discarded during input by the read routine. Likewise, every occurrence of the letterizer character as a data character in a character string must be prefixed by the letterizer character.

For example, the string

```
'|''
```

contains one character (a string delimiter), and the string

```
'|||''
```

contains two characters (a letterizer and a string delimiter).

When it is not necessary to represent a character string whose total capacity is larger than the shortest vector necessary to contain the characters specified, the simpler form:

```
'c...c'
```

may be used. This designates a character vector which may have zero, one, two or three unused elements. Referring to format diagram, it may be seen that if N is the number of real characters in a string (letterizing characters are not counted), the number of unused elements for this simplified notation is the residue, (N-1) MOD 4.

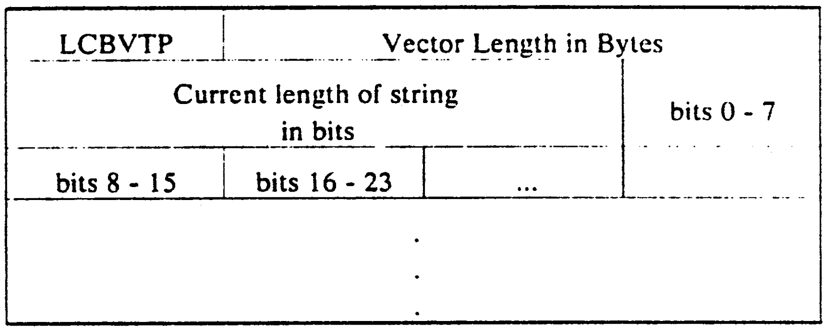
Example: to specify an eight-element character vector containing the letters 'FUNCTION', write:

```
'FUNCTION'
```

This vector will have space for nine characters (see Format diagram.) and a current length of eight. To specify a vector with a capacity of 100 characters, but with a current length of zero, write:

```
%100''
```

Bit Vector Format:



Pointer type class: *SB*. A distinguished primitive class.

Abstract syntax: $SB\{M\} = \{\textbf{truth-value}^{n}\}$

Primitives:

Type predicate: %•SB

The type constraint: %•=SB
(%•=SB ptr_1) = ptr_1 iff $ptr_1 = \epsilon\ SB$, otherwise ∇.

The bit string allocator:

%•SBGET : $SMI \times M \rightarrow SB \times M$
(%•SBGET smi_1) $\{M_1\} = sb_1\{M_2\}$ = %Bk[:m]''
k = (((%•MOD (smi_1+ (31+(3×8))) 32) × 32) − 24)
$M_2 \sim M_1 = sb_1\{M_2\}$
$sb_1 \notin PTR$ of M_1
Allocates a bit string with capacity for at least smi_1 bits.

The input/output format of bit vectors is similar to the format for character vectors; however; 4 bit elements are represented by one hex character and the current length field is a count of the number of bits in the vector, not a count of the number of bytes (see Format diagram.). Only the characters 0...9 and A...F may be specified as part of a bit string.

There are different input/output representations for bit vectors, depending upon the current length of the vector being considered. For bit vectors whose length is a multiple of four bits, the format is:

%Bk'h...h'

where 'k' is the maximum number of bits which the specified vector could contain. The actual contents of the bit string 'h...h' reflects only the current length of the string, and might be null.

As with character vectors, the maximum length field is optional and may be omitted when representing a vector of length consistent with the explicitly specified data. A bit vector specified without an explicit maximum length 'k' and with up to 28 unused elements has the format:

%B'h...h'

For bit vectors whose current length is not a multiple of four bits, the format is:

%Bk:c'h...h'

where 'k' is as previously defined and c is the current number of bits in the string. A bit vector specified without a maximum 'k', but with a current length 'c' and with up to 31 unused elements has the format:

%B:c'h...h'

RECORDS

Records are objects whose components are not necessarily of all the same type.

Pointer type class: $REC = \{ID \cup APPL \cup NT \cup PLEX\}$

Abstract syntax: $REC\{M\}$ = { **identifier** ∪ **applicative-object** ∪ **n-tuple** ∪ **complex** }

Primitives:

Type predicate: %•REC

The type constraint: %•=REC

(%•=REC ptr_1) = ptr_1 iff $ptr_1 = \epsilon\ REC$, otherwise ∇.

IDENTIFIERS

Identifiers are objects that have a **pname** component. Identifiers are used as the names of variables in e the expression interpretation of **s-exp**. In the past, the world of LISP data objects was divided into pairs and atoms. The numbers were a distinguished (reserved) subrange, as was NIL, what are now *SF* and *UR*, etc. Over the years the class *ATOM* has come to mean, "not a pair", a somewhat miss-named distinction. As a result of the elaboration of the type schema, it is no longer necessary or desirable to have the evaluator consider any subrange of the identifiers as reserved.

An important characteristic of an identifier is whether it is %•INTERNed or not, and if it is then in what **obarrays**.

The **normal-identifiers** are those which are %•INTERNed in the distinguished system obarray **OBARRAY**, and only therein. For such identifiers, called *NORID*, %•READ preserves %•EQ-ness.

Gensyms on the other hand are never interned and %•READ preserves only local %•EQ-ness when they are recognized. In other words new ones are allocated.

The **uninterned-identifiers** and the **multiply-interned-identifiers** are not yet available and will not be precisely defined.

Pointer type class: $ID = \{NORID \cup GENSYM \cup UNID\dagger \cup MOBID\dagger\}$

Abstract syntax: $ID\{M\}$ = { **normal-identifier** ∪ **generated-symbol** ∪
uninterned-identifier†∪ **multiply-interned-identifier**†}

Primitives:

Type predicate: %•ID

The type constraint: %•=ID

(%•=ID ptr_1) = ptr_1 iff $ptr_1 = \in ID$, otherwise ∇.

The PNAM-property function:

%•PNAM : $ID \times M \rightarrow \{STR \mid SMI \mid SMI \times STR \mid STR \times LIST\} \times M$

Case 1: $NORID \rightarrow STR$

(%•PNAM $norid_1$) = str_1,

where $str_1\{M\}$ = '$char_0$...', and

where $norid_1\{M\} = char_0\ldots$, and

Case 2: $GENSYM \rightarrow SMI$

(%•PNAM $gensym_1$) = smi_1,

where $gensym_1\{M\}$ = %Gsmi_1

Case 3: $UNID \rightarrow SMI \times STR$ †

```
        (%•PNAM $unid_1$) = $pr_1$ ,
            where $pr_1$ = ( $smi_1$ • $str_1$),
                where $str_1\{M\}$ = '$char_0$...' , and
                where $unid_1\{M\}$ = %$smi_1$:$char_0$... .
    Case 4: $MOBID \rightarrow STR \times LIST$ †
        (%•PNAM $mobid_1$) = $pr_1$ ,
            where $pr_1$ = ( $str_1$ • $list_1$),
                where $str_1\{M\}$ = '$char_0$...' , and
                where $unid_1\{M\}$ = %•($char_0$... • $list_1$).
```

The *NORID* allocator:

%•INTERN : $SC \times M \rightarrow NORID \times M$

(%•INTERN str_1)$\{M_1\}$ = $norid_1\{M_2\}$

Where if an element of the global object array whose PNAME component is %•EQUAL to the argument string *sc*, then the resultant value is %•EQ to that object. If on the other hand no element of the global object array with the same print name is found then a new **norid** is allocated, and the global object array is updated to include it.

The canonical print representation of a **norid** is simply the characters of its pnam except that certain of those characters must be letterized. For instance: any initial character that is a **digit**, or any **id-delimiter** character.

The *GENSYM* allocator:

%•GENSYM : $M \rightarrow GENSYM \times M$

(%•GENSYM) $\{M_l\}$ = $gensym_l\{M_2\}$

$M_2 \sim M_l = gensym_l\{M_2\}$

$gensym_l \notin PTR$ of M_l

The canonical print format for a given $gensyms_l$ is:

%G*gennum* where *gennum* is the print form of (%•PNAM $gensym_l$).

APPLICATIVE-OBJECTS

In describing the semantics of applicative objects, the relevant sections of the meta-linguistic formal description section are referenced by the subsection numbers of that section. The reader can refer to that section for the detailed description of their the semantics.

Pointer type class: $APPL = \{ABST \cup SD \cup FUN \cup BPI \cup SF \cup UR\}$

Abstract syntax: $APPL\{M\}$ = { **abstraction** U **state-descriptor** U **funarg** U **binary-program-image** U **special-function** U **understood-rator**}

Primitives:

Type predicate: %•APPL

The type constraint: %•=APPL

(%•=APPL ptr_1) = ptr_1 iff ptr_1 = ϵ *APPL*, otherwise ∇.

ABSTRACTIONS

This class of constant objects that are applicative is the consequence of eschewing reserved combination forms. We found the need for an anonymous self describing computational object that would play the role that was formerly played by things like lambda-expressions. The binary program objects fill this role but are practically unutterable. We needed an object which had expressions as components.

Pointer type class: *ABST* = {*LAM* ∪ *MLAM* ∪ *ORCD*}

Abstract syntax: *ABST{M}* = { **lambda-abstraction** ∪ **mlambda-abstraction** ∪ **mu-abstraction** ∪ **operator-code-abstraction**}

Primitives:

Type predicate: %•ABST

The type constraint: %•=ABST

(%•=ABST ptr_1) = ptr_1 iff ptr_1 = ϵ *ABST*, otherwise ∇.

LAMBDA ABSTRACTIONS

Lambda-abstractions are ordinary applicative ("anonymous") function description constants. The description consists of two component parts:

1. The **bv** part describes the nature of the list of argument values and the variables which conform to the components of this list.
2. The **exp-seq** part, taken in the context of the function and its parameter variables, denotes the value of the function.

Pointer type class: *LAM* a distinguished type class.

Abstract syntax: *LAM{M}* = **lambda-abstraction** = **bv** × **exp-seq**

Primitives:

Type predicate: %•LAM

The type constraint: %•=LAM

$(\%\bullet\!=\!\mathrm{LAM}\ ptr_1) = ptr_1$ iff $ptr_1 = \epsilon\ LAM$, otherwise ∇.

Access functions: %•BV and %•ESEQ.

$\%\bullet\mathrm{BV} : LAM \rightarrow PTR$
$(\%\bullet\mathrm{BV}\ lam_l) = ptr_l$
where $lam_l = \%(\%\text{,LAMBDA}\ ptr_l \bullet ptr_2)$

$\%\bullet\mathrm{ESEQ} : LAM \rightarrow PTR$
$(\%\bullet\mathrm{ESEQ}\ lam_l) = ptr_2$
where $lam_l = \%(\%\text{,LAMBDA}\ ptr_l \bullet ptr_2)$

Allocation function:

$\%\bullet\mathrm{LAMGET} : PTR \times PTR \times M \rightarrow LAM \times M$
$(\%\bullet\mathrm{LAMGET}\ ptr_1\ ptr_2)\ \{M_1\} = lam_3\ \{M_2\} = \%(\%\text{,LAMBDA}\ ptr_1 \bullet ptr_2)$
where $lam_3 \notin PTR$ of M_1, and $lam_3 \in PTR$ of M_2, or simply, a new pointer is allocated.
$(\%\bullet\mathrm{BV}\ lam_3)\ \{M_2\} = ptr_1$
$(\%\bullet\mathrm{ESEQ}\ lam_3)\ \{M_2\} = ptr_2$

Print form:
%(%,LAMBDA ***bv*** • ***exp-seq***)
where ***bv*** the print representation of the bound-variable part is
{***c*** | (FLUID ***iden***) | (LEX ***iden***) | ***iden*** | ($bv_l \bullet bv_2$) }
where ***iden*** is ([%•=*type-class*] ***id***)
and ***exp-seq*** is {***atom*** | (***e***...)}

Semantics: See rules 8.1.1.1. and 11.2.

The question has come up as to why an explicit lambda-abstraction in operator is not lexical. As in:
(%(%,LAMBDA (X) (%•CONS X Y)) Y)
The answer is that it denotes itself and not a closure. A lambda-expression however denotes a closure that captures the current environment and that includes lexicals. The fact that the semantics describes several closure forming avoidance features is largely a matter of pragmatics.

A similar argument holds for a quoted lambda expression which denotes a lambda expression not a closure.

MLAMBDA ABSTRACTIONS

Mlambda-abstractions are macro-composition (macro) description constants. Macro composition is a transformation from an operator value which is a macro and the original combination's data structure, (*rator rand...*), which produces a new expression. The description consists of two parts:

1. The **bv** part describes the nature of the argument and the variables which conform to the components of this list.
2. The **exp-seq** part, taken in the context of the macro and its parameter variables, denotes the value of the function.

Pointer type class: *MLAM* a distinguished type class.

Abstract syntax: *MLAM{M}* = **mlambda-abstraction** = **bv** × **exp-seq**

Primitives:

Prof. Dr. H. Stoyan
Universität Erlangen-Nürnberg
Institut für Mathematische Maschinen
und Datenverarbeitung (Informatik VIII)
Am Weichselgarten 9
91058 Erlangen

Type predicate: %•MLAM

The type constraint: %•=MLAM

(%•=MLAM ptr_1) = ptr_1 iff ptr_1 = ∊ *MLAM*, otherwise ∇.

Access functions: %•MBV and %•MESEQ.

%•MBV : *MLAM* → *PTR*
(%•MBV $mlam_l$) = ptr_l
where $mlam_l$ = %(%›MLAMBDA ptr_l • ptr_2)

%•MESEQ : *MLAM* → *PTR*
(%•MESEQ $mlam_l$) = ptr_2
where $mlam_l$ = %(%›MLAMBDA ptr_l • ptr_2)

Allocation function:

%•MLAMGET : *PTR* × *PTR* × *M* → *MLAM* × *M*
(%•MLAMGET ptr_1 ptr_2) $\{M_1\}$ = $mlam_3$ $\{M_2\}$ = %(%›MLAMBDA ptr_1 • ptr_2)
where $mlam_3$ ∉ *PTR* of M_1, and $mlam_3$ ∊ *PTR* of M_2, or simply, a new pointer is allocated.
(%•MBV $mlam_3$) $\{M_2\}$ = ptr_1
(%•MESEQ $mlam_3$) $\{M_2\}$ = ptr_2

Print form:
%(%›MLAMBDA ***bv*** • ***exp-seq***)
where ***bv*** the print representation of the bound-variable part is

{c | (FLUID ***iden***) | (LEX ***iden***) | ***iden*** | (bv_1 • bv_2) }
and ***exp-seq*** is {***atom*** | (***e***...)}

Semantics: See rules 8.1.1.2 and 9.2.

MU ABSTRACTIONS

Mu-abstractions are context description constants. They are special form applicable to the unevaluated list of operand expressions and result in an ordinary application. The description consists of two parts:

1. The **bv** part describes the nature of the argument parameter list and the variables which conform to the components of this list.
2. The **value-list** part, the parameter values.

Pointer type class: MU a distinguished type ciass.

Abstract syntax: $MU\{M\}$ = **mu-abstraction = bv × s-exp**

Primitives:

Type predicate: %•MU

The type constraint: %•=MU

(%•=MU ptr_1) = ptr_1 iff $ptr_1 = \epsilon\ MU$, otherwise ∇.

Access functions: %•MBV and %•MUVAL.

%•MBV : $MU \rightarrow PTR$
(%•MBV mu_1) = ptr_1
where mu_1 = %(%•MU ptr_1 • ptr_2)

%•MUVAL : $MU \rightarrow PTR$
(%•MUVAL mu_1) = ptr_2
where mu_1 = %(%•MU ptr_1 • ptr_2)

Allocation function:

%•MUGET : $PTR \times PTR \times M \rightarrow MU \times M$
(%•MUGET ptr_1 ptr_2) $\{M_1\}$ = mu_3 $\{M_2\}$ = %(%•MU ptr_1 • ptr_2)
where $mu_3 \notin PTR$ of M_1, and $mu_3 \in PTR$ of M_2, or simply, a new pointer is allocated.
(%•MBV mu_3) $\{M_2\}$ = ptr_1
(%•MUVAL mu_3) $\{M_2\}$ = ptr_2

Print form:

%(%›MU ***bv*** • ***value-list***)

where ***bv*** the print representation of the bound-variable part is

{***c*** | (FLUID ***iden***) | (LEX ***iden***) | ***iden*** | ($\boldsymbol{bv_1}$ • $\boldsymbol{bv_2}$) }

and ***value-list*** is an ***s-exp***

Semantics: See rules 8.3 and 13.11.

SEQUENCE ABSTRACTIONS

Sequence-abstractions are ordinary applicable much like **lambda-abstraction**. They differ in that ***E*** is not changed and no new activation record is created. The description consists of three parts:

1. The **tag** part names this sequence so that exit-expressions can be sequence specific.
2. The **aux** part, analogous to **bv** but creates references to stack places rather than **bindings** of ***E***.
3. The list of program statements **ps-list**.

Pointer type class: *SEQ* a distinguished type class.

Abstract syntax: *SEQ{M}* = **sequence-abstraction = tag × aux × ps-list**

Primitives:

Type predicate: %•SEQ

The type constraint: %•=SEQ

(%•=SEQ ptr_1) = ptr_1 iff $ptr_1 = \epsilon$ *SEQ*, otherwise ∇.

Access functions: %•TAG, %•AUX and %•PSLST.

%•TAG : *SEQ* → *PTR*

(%•TAG *seq-abstraction*$_I$) = *tag*$_I$

where *seq-abstraction*$_I$ = %(%›SEQ *tag*$_I$ *aux s* ...)

%•AUX : *SEQ* → *PTR*

(%•AUX *seq-abstraction*$_I$) = *aux*$_I$

where *seq-abstraction*$_I$ = %(%›SEQ *tag aux*$_I$ *s* ...)

%•PSLIST : *SEQ* → *PTR*

(%•PSLIST *seq-abstraction*$_I$) = *ps-list*$_I$

where *seq-abstraction*$_I$ = %(%›SEQ *tag aux* • *ps-list*$_I$)

Allocation function:

%•SEQGET : $\{IDU()\} \times PTR \times PTR \times M \rightarrow SEQ \times M$

(%•SEQGET tag_1 aux_1 $ps\text{-}list_1$) $\{M_1\}$ = $seq\text{-}abstraction_3$ $\{M_2\}$ = %(%•SEQ tag_1 aux_1 • $ps\text{-}list_1$)

where $seq\text{-}abstraction_3 \notin PTR$ of M_1, and $seq\text{-}abstraction_3 \in PTR$ of M_2, or simply, a new pointer is allocated.

(%•TAG $seq\text{-}abstraction_3$) $\{M_2\} = tag_1$

(%•AUX $seq\text{-}abstraction_3$) $\{M_2\} = aux_1$

(%•PSLIST $seq\text{-}abstraction_3$) $\{M_2\} = ps\text{-}list_1$

Print form:

%(%•SEQ ***tag aux • ps-list***)

Semantics: See rules 8.1.1.5, 11.9, 11.10 and 16.

OPERATOR CODE ABSTRACTIONS

Operator-code-abstractions are a bit odd; from the point of view of the interpreted semantics it is just an operator expression with a great deal of redundant information attached, compilation is determined by this information and the other operator expression is ignored. The description consists of three parts:

1. The **rator** part is an expression that the interpreter considers to be equivalent to the abstraction itself.
2. The **f-list** part, alerts the compiler to the free variables required.
3. The list **lap-code** of assembly code statements for the LAP assembler.

Pointer type class: *FRCODE* a distinguished type class.

Abstract syntax: *FRCODE{M}* = **operator-code-abstraction** = **rator** × **f-list** × **lap-code**

Primitives:

Type predicate: %•FRCODE

The type constraint: %•=FRCODE

(%•=FRCODE ptr_1) = ptr_1 iff $ptr_1 = \in FRCODE$, otherwise ∇.

Access functions: %•RATOR, %•FLIST and %•LAPCODE.

%•RATOR : $FRCODE \rightarrow PTR$

(%•RATOR $operator\text{-}code\text{-}abstraction_1$) = $rator_1$

where $operator\text{-}code\text{-}abstraction_1$ = %(%•FR*CODE $rator_1$ $f\text{-}list$ • $lap\text{-}code$)

%•FLIST : *FRCODE* → *PTR*
(%•FLIST *operator-code-abstraction*$_i$) = *f-list*$_i$
where *operator-code-abstraction*$_i$ = %(%›FR*CODE *rator f-list*$_i$ • *lap-code*)

%•LAPCODE : *FRCODE* → *PTR*
(%•LAPCODE *operator-code-abstraction*$_i$) = *ps-list*$_i$
where *operator-code-abstraction*$_i$ = %(%›FRCODE *rator f-list* • *ps-list*$_i$)

Allocation function:

%•FRCODEGET : *PTR* × *PTR* × *PTR* × *M* → *FRCODE* × *M*
(%•FRCODEGET *rator*$_1$ *f-list*$_1$ *ps-list*$_1$) $\{M_1\}$ = *operator-code-abstraction*$_3$
$\{M_2\}$ = %(%›FR*CODE *rator*$_1$ *f-list*$_1$ • *ps-list*$_1$)
where *operator-code-abstraction*$_3$ $\notin$ *PTR* of M_1, and
operator-code-abstraction$_3$ $\in$ *PTR* of M_2, or simply, a new pointer is allocated.
(%•RATOR *operator-code-abstraction*$_3$) $\{M_2\}$ = *rator*$_1$
(%•FLIST *operator-code-abstraction*$_3$) $\{M_2\}$ = *f-list*$_1$
(%•LAPCODE *operator-code-abstraction*$_3$) $\{M_2\}$ = *lap-code*$_1$

Print form:
%(%›FRCODE ***rator f-list* • *lap-code***)

Semantics: See rules 8.1.1.4, and 11.8.

STATE DESCRIPTOR

State descriptors serve three purposes. Firstly, they define an environment and are used as a component of **funargs** (closures) for that purpose.

Secondly, they are actually saved states which may be applied to effect an leaving of the current state and the continuation of the saved state. Execution will subsequently proceed in the environment of the saved state, at the point immediately following the STATE operation which created the saved state.

Thirdly, they are used for the implementation of binding search avoidance trick. A special metalinguistic component is added to denote the current environment-path. Environment-path identifiers are metalinguistic data objects whose principle property is that they identify an environment search path. A secondary, but useful, property is that they possess some space for saving and restoring some state components during path switching. As a result of much consideration, several false starts, and dogged persistence, the ideal embodyment of environment-path identifiers is believed to be: state descriptors. These environment path descriptions are used in conjunction with the shallow binding cells to avoid searching *E* in many cases. The total state then consists of the ordinary state, now shown to be $\{S;E;C;D;X\}$, applied to *M*, applied to the environment path identifier.

i.e. $\{S;E;C;D;X\}$ $\{M\}$ $\{sd\}$.

It may prove to be a pragmatic necessity to create another object just for purposes one and three. In these cases only ***E*** need be retained. The nature of our current implementation is such that ***E*** is not independent of ***D*** so no benefit would be realized. We are waiting for our experience with this model to provide some guidance.

Refer to the section on Global Environments for a description of *sd* and its components.

Pointer type class: *SD* a distinguished primitive class.

Abstract syntax: *SD{M}* = **state-descriptor** = {***D;sd;X;gloE***}.

Type predicate: %•SD

The type constraint: %•=SD

(%•=SD ptr_1) = ptr_1 iff $ptr_1 = \epsilon$ *SD*, otherwise ∇.

The state allocation (or saving) operator: %:STATE

See rule 11.4.2.2.

Two state descriptors are %•EQUAL or %•EQUUP iff they are %•EQ. The reason is that we currently lack the motivation to descend the structure. The same is true for %•READ and %•PRINT. If the meta-linguistic states, which occur as components of state descriptors, were themselves data objects then it would be imperative that they be first class. For reasons of stack deletion strategy this alternative was vetoed. Perhaps, the future holds promise of efficient, meta-linguistic states implemented as first class data objects.

Semantics:

State-descriptors evaluate as constants but the application of one is understood and of the class of ordinary applications (*rands* evaluated).

See rule 11.7.

FUNARGS

A **funarg** is an expression closure -- that is, the combination of an expression with a specific environment, contained in a **sd** component.

It has the following representation:
%(%•FUNARG *e* • *sd*)

Note: %›FUNARG is not an applicative constant. It is merely part of the special bracket symbol "%(%›FUNARG". This contrasts with %›LAMBDA and %›MLAMBDA which are applicative and are also used to form special bracket symbols.

Funarg's have been called function closures because of the interpretation placed upon these objects when they appear in particular contexts, such as an operator. It is more correct to think of them as expression closures.

Funargs (closures) are closed in the following computational sense: the bindings of the free variables are fixed (closed) but because those bindings are updateable the meaning is not closed until evaluation. In other words, the closure contains the information about where resides the values upon which the meaning depends.

Pointer type class: *FUN* a distinguished primitive class.

Abstract syntax: $FUN\{M\}$ = **funarg** = **expression** $\times$ E

Semantics: See rules 6, 8.2, 9.3, 11.6, 13.12, 13.13, and 14.

Primitives:

Closure forming primitives: {%›FUNCTION | %›LAMBDA | %›MU | %›MLAMBDA | %›FR*CODE}
See *SF* application rule 13.7 and rule 14.

Type predicate: %•FUN

The type constraint: %•=FUN

(%•=FUN ptr_1) = ptr_1 iff ptr_1 = ϵ FUN, otherwise ∇.

Funargs suffer from the fact that they contain a state descriptor as a component. Therefore, they are not first class values objects. %•READ, %•PRINT, and %•EQUAL are not well behaved with respect to them. Once again complete consistency has been missed due to lack of motivation.

BINARY PROGRAM IMAGES

Bpi object are applicative objects that are executed directly by the hardware interpreter as opposed to **abstractions** which are LISP interpretable. These objects are usually the result of compiling abstractions. In such a case it is the compilers responsibility that they be well formed. The actual case is that because of LAP and the %(%›FR*CODE construct a **bpi** which is ill behaved may be formed.

Once again we have the intrusion of system programmer activity into the "sanctity" of an otherwise inviolable system.

Pointer type class: $BPI = \{FBPI \cup MBPI\}$

Abstract syntax: $BPI\{M\}$ = { **function-binary-program** ∪ **meta-program** }

Primitives:

Type predicate: %•BPI

The type constraint: %•=BPI

(%•=BPI ptr_1) = ptr_1 iff $ptr_1 = \in BPI$, otherwise ∇.

It is not yet possible to print binary program images in a form which would permit them to be subsequently read by LISP and used like the original object. There are several reasons for this, the major difficulty being the lack of interest due to the availability of a package of special purpose programs for this purpose alone.

Therefore, since it frequently occurs that an object being printed contains references to binary programs (e.g. in a backtrace), a convention is used which incorporates the identification message of a binary program (normally, the identifier associated with the BPI when it was compiled) in the form:

%F'BPI*message*' or %M'BPI*message*'

where F is used for functions with evaluated arguments, and M is used for meta-applicative binary-programs.

If an attempt is made to read such a form, the read program will emit an error message and use the .NOVAL object instead of a binary program.

FUNCTION BINARY PROGRAM IMAGES

Pointer type class: *FBPI* = a distinguished primitive type.

Abstract syntax: $FBPI\{M\}$ = **function-binary-program**

Primitives:

Type predicate: %•FBPI

The type constraint: %•=FBPI

(%•=FBPI ptr_1) = ptr_1 iff ptr_1 = ϵ *FBPI*, otherwise ∇.

Semantics: See rules 11.3, 11.4.2.1.1, and 11.6.

META PROGRAMS

Pointer type class: *MBPI* = a distinguished primitive type.

Abstract syntax: *MBPI{M}* = **meta-program**

Primitives:

Type predicate: %•MBPI

The type constraint: %•=MBPI

(%•=MBPI ptr_1) = ptr_1 iff ptr_1 = ϵ *MBPI*, otherwise ∇.

Semantics: See rules 8.2, 9.1, 9.3.1, and 11.5.

SPECIAL FORMS

The **special-forms** are a distinguished class of constants that apply specially. In most other LISP systems certain combination forms are reserved for the purpose of these special constructs.

Pointer type class: *SF* a distinguished primitive class.

Abstract syntax: *SF{M}* = { %,LAMBDA | %,MLAMBDA | %,QUOTE | %,SETQ | %,FUNCTION | %,LABEL | %,COND | %,SEQ | %,GO | %,EXIT | %,PROGN | %,RETURN | %,FR*CODE }

Primitives:

Type predicate: %•SF

The type constraint: %•=SF

(%•=SF ptr_1) = ptr_1 iff $ptr_1 = \epsilon$ *SF*, otherwise ∇.

The following constants (*sf*) occur as *rator* value and denote special forms, i.e. their application is special and defined by special rules involving transformations of the metalinguistic machine.

{ %,LAMBDA | %,MLAMBDA | %,QUOTE | %,SETQ | %,FUNCTION | %,LABEL | %,COND | %,SEQ | %,GO | %,EXIT | %,PROGN | %,RETURN | %,FR*CODE }

As such special forms apply specially i.e., they are applied to their unevaluated *randlist*, they often require that *randlist* have a definite syntax. The required syntax for these built-in operators was defined earlier.

Special Forms Application

See rules 8.1.3, 8.3, 11.11, and 13.

UNDERSTOOD OPERATORS

Pointer type class: *UR* = {*FIX-UR* ∪ *MULT-UR*}

Abstract syntax: *UR{M}* = {**fix-ur** ∪ **mult-ur**}

Primitives:

Type predicate: %•UR

The type constraint: %•=UR

$(\%\bullet{=}\mathrm{UR}\ ptr_1) = ptr_1$ iff $ptr_1 = \epsilon\ UR$, otherwise ∇.

The understood operator class of applicative objects all are ordinary applicative (they receive their arguments evaluated). The two subclasses distinguish those which have a fixed number of operands **fix-ur** from those that have multiple operands **mult-ur**.

FIX-UR

Pointer type class: *FIX-UR*

Abstract syntax: *FIX-UR{M}* = **fix-ur**

Primitives:

Type predicate: %•FIX-UR

The type constraint: %•=FIX-UR

$(\%\bullet{=}\mathrm{FIX\text{-}UR}\ ptr_1) = ptr_1$ iff $ptr_1 = \epsilon\ FIX\text{-}UR$, otherwise ∇.

The print representation for this class is %• followed by a reserved name. The semantics of applying these constants is built in and the definitions have been previously given. Several are sufficiently special to merit comment, namely:

%•APPLX, %•EVAL, %•EVA1, and %•SET.

(%•APPLX *fn list*) (RULE 11.4.1.3)

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 not accessible during this application.

(%•MDEFX *fn form*) (RULE 11.4.1.2.)

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*) (RULE 11.4.1.1.)

EVA1 evaluates its one operand value with respect to the current environment. Lexical variables are not accessible during this evaluation.

(%•EVAL *e sd*) (RULE 11.4.1.4.)

EVAL evaluates its first operand value with respect to the context of the state which is the value of its second operand. This operator is very significant because it, along with state descriptors and fluid variables, gives LISP its ability to dynamically construct an expression and then evaluate it with respect to an independent context. Were these not present the environment and control would march along in lockstep and retention strategy would not be required.

(%•SET a_1 a_2) (RULE 19.)

SET is like SETQ except it evaluates its first operand, which must have an identifier as value. Lexical variables are not accessible for this assignment.

MULT-UR

Pointer type class: *MULT-UR*

Abstract syntax: *MULT-UR{M}* = **mult-ur**

Primitives:

Type predicate: %•MULT-UR

The type constraint: %•=MULT-UR

(%•=MULT-UR ptr_1) = ptr_1 iff ptr_1 = ϵ *MULT-UR*, otherwise ∇.

The print representation for this class is %: followed by a reserved name. The semantics of applying these constants is built in and the definitions have been previously given. Several are sufficiently special to merit comment, namely:

%:CALL, and %:STATE.

(%:CALL a_1 ... *fn*) (RULE 11.4.2.1.)

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.

(%:STATE [*gloval* [*glolst*]]) (RULE 11.4.2.2.)

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***.

NTUPLES

Ntuples are provided in the hope that a type extension method will use them. Because of this they ought not to be used except through this facility which has yet to be defined. The system is prepared to storage manage, print and read them. They are mentioned only as an inducement.

Ntuple Format:

LCMVTP	Vector Length in Bytes
Small integer length in bytes of pointer section	
Pointer for Element 0	
Pointer for Element 1	
. . .	
Pointer for Last Element	
Unstructured binary data which is accessible only via a user-written function. . . .	

Pointer type class: *NT*

Abstract syntax: *NT{M}* = **ntuple**

Primitives:

Type predicate: %•NT

The type constraint: %•=NT

$(\%\bullet{=}\mathrm{NT}\ ptr_1) = ptr_1$ iff $ptr_1 = \epsilon\ NT$, otherwise ∇.

Ntuple structures present a difficult problem for printing, because there is no standard organization of the binary data section. Therefore, the print representation of an ntuple is:

%(• **comp*** • **bitstring)**

In effect, the binary data part of a selector structure is printed as if it were a bit vector.

COMPLEXES

Complexes are provided in the hope that a type extension method will use them. Because of this they ought not to be used except through this facility which has yet to be defined. The system is prepared to storage manage, print and read them. They are mentioned only as an inducement.

Complexes Format:

LCMVTP	Vector Length in Bytes
Small integer length in bytes of pointer section	
Pointer for Element 0	
Pointer for Element 1	
. . .	
Pointer for Last Element	
Unstructured binary data which is accessible only via a user-written function. . . .	

Pointer type class: *PLEX*

Abstract syntax: *PLEX{M}* =

Primitives:

Type predicate: %•PLEX

The type constraint: %•=PLEX

(%•=PLEX ptr_1) = ptr_1 iff ptr_1 = ϵ *PLEX*, otherwise ∇.

Complexes present a problem for printing, because there is no standard organization of the binary data section of a complex. Therefore, the print representation of a complex is:

```
%(•• comp' • bitstring)
```

In effect, the binary data part of a complex is printed as if it were a bit vector.

ISSUES and COMMENTS

The introduction of the *nlambda-abstraction* class of operators and their attendant macro composition forms leads to a more complex formal definition than is usual for LISP. It seems important, however, to raise these long term denizens of LISP systems to first class status. This choice also leads to the strategy of evaluating the operator once and classifying the type of application on the basis of that value.

The treatment of special forms and understood basic operators will be seen as considerably different than the usual practice. While this treatment requires a few special classes of constants, it gives back the full set of identifiers for use as variables.

The environment was admittedly embellished to provide a model for lexical bindings. This gives rise to distinguished contours and coincidentally distinguished states.

The environment model was also extended to encompass the notion of variables of constrained type. So long as we persist in the belief that we should be able to move freely from compiled to interpretive evaluation or that the basic model for meaning is the interpreter, then we feel obligated to have interpreter models for concepts even if they arise from compilation technology. Note also the CALL construct.

The notion of dynamic evaluation context was continued in this LISP, the notion of capturing a context and retaining it for later use was preserved. The **fluid-variables** comprise the dynamicly inherited environment. We have required that they be distinguished whereas the **lexical-variables** are obtained by default.

The treatment of global environments is thought to be a reasonable extension and improvement over "atom-head-bindings".

The MU operator has been provided with the ability to expose lexicals (somewhat grudgingly) as a powerful system programming tool. The ability to describe contexts abstractly is due largely to Fraser[15]. This concept could have been realized by a method that was conservative with regard to lexical access. The writers of the compiler and the debugging facilities insisted on the right to implement these facilities entirely in LISP. A conclusion of this was to provide a window into lexicals.

The SEQ operator represents considerable evolution in this design, deriving from the PROG form. Statement sequences do not create binding contours, they can however create named stack places. Perhaps we have intruded our desire to illustrate a computational consideration. The addition of the *tag* component was seen as the solution to a problem that arose when sequences were automatically being wrapped around expressions by macro's. Wrapping gave rise to misinterpretations for exit expressions that were imbedded within wrapped expressions.

Expression sequences we though to be sufficiently different from statement sequences to merit special treatment. We note that they are parsimonious in the use of ***S***.

We also considered the so called implied PROGN for abstractions and conditionals as a generally good idea.

The model described above treats GO as strictly local to a statement context. This was done with considerable malice of forethought because it makes for a simpler semantics. The unique behavior of this operator with respect to the state (i.e. it only affects the stack and control) has led to its inclusion as a primitive. The semantics for go expressions are somewhat complicated by the possibility that they can occur anywhere. This small complexity does not, however, preclude a simple, efficient, compiled realization, namely change of location counter.

The powerful operator STATE and the *sd* data objects were introduced in order to model complex control structures. There is reason to question whether certain control construct are not deserving enough to merit direct, computationally efficient, primitive status.

Streams and the interrupt schema must be considered as not completed. As the current models are used they probably will develop and be revised.

The treatment of self-referring structures, particularly with regard to the output canonical representation and its relationship to equality is though to be a bit more thorough that in most LISP systems.

LISP1.8+0.3i provides a rich but rather ad hoc and fixed set of data objects. A very general data type extension was anticipated (using **ntuples** and **complexes**) but has never been completed.

For all the issues, lacks, and controversy, this effort at detailed definition has proved to be of some benefit, as a specification document, to the designers and it is hope that it will be even helpful to implementors.

LISP1.8+0.3i is a result of our developing LISP/370 and it represents how we would propose to do it if we were to build another LISP system. Actually it might be better to say that there are several proposals in our local community and this is one.

In conclusion, to repeat, the purpose of this document is to encourage interest and comment. The author welcomes any and all responces and commends those with the persistence to have read any large part of this wearying document.

Acknowledgements

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

APPENDIX A

Lexicon of named states.

$D_{non\text{-}conformal\text{-}app}$ =

{x•a$_n$•...a$_1$•S_l;*E*;
(APP$_l$•S$_l$)•()• %(<u>LAMBDA</u> ?ARGS? ((ERR2 4)?ARGS?))•*C*;*D*} .

$D_{macro\text{-}non\text{-}conformal}$ =

{m•(m *rand*...)•*S*;*(APP$_l$•S)* •()•%(<u>LAMBDA</u> ?ARGS? ((ERR2 3) ?ARGS?))•*C*;*D*}

$D_{macro\text{-}inapplicable}$ =

{x•a$_n$•...a$_1$•S_l; *E*;
(APP$_l$•S$_l$)•()• %(<u>LAMBDA</u> ?ARGS? ((ERR2 5)?ARGS?))•*C*; *D*}

$D_{inapplicable\text{-}object}$ =

{x•a$_n$•...a$_1$•S_l; *E*;
(APP$_l$•S$_l$)•()• %(<u>LAMBDA</u> ?ARGS? ((ERR2 6)?ARGS?))•*C*; *D*}

$D_{unbound\text{-}aux}$ =

{a$_1$•*S*; *E*;
(APP$_l$•S)•()• %(<u>LAMBDA</u> ?ARGS? ((ERR2 15)?ARGS?))•*C*; *D*}

$D_{ill\text{-}formed}$ =

{x•a$_1$•*S*; *E*;
(APP$_l$•S)•()• %(<u>LAMBDA</u> ?ARGS? ((ERR2 16)?ARGS?))•*C*; *D*}

$D_{exit\text{-}error}$ =

{x•a$_1$•*S*; *E*;
(APP$_l$•S)•()• %(<u>LAMBDA</u> ?ARGS? ((ERR2 17)?ARGS?))•*()*; *D*}

$D_{unbound\text{-}AUXSET}$ =

{z•id_l•*S*; *E*;
    (APP$_l$•S)•()• %(<u>LAMBDA</u> ?ARGS? ((ERR2 18)?ARGS?))•*C*; *D*}

References

[1] Backus, J. *Reduction languages and variable-free programming*. IBM Research report RJ1010, Yorktown Heights, N.Y., April 7, 1972

[2] Backus, J. *Programming languages semantics and closed applicative languages.* IBM Research Report RJ1245

[3] Bekic', H., Walk, K. Formalization of storage properties. Springer-Verlag, *Symposium on semantics of algorithmic languages*. Lecture notes in mathematics No. 188 pp. 28-61 (1971).

[4] Bobrow, D. G., Wegbreit, B. *A Model and Stack implementation of Multiple Environments*. CACM Vol. 16, No. 10 pp 591-612 1975.

[5] Gordon, M. J. C. *Models of pure LISP*. Ph.D. Thesis. University of Edinburgh. (1973)

[6] Landin, P. J. *The mechanical evaluation of expressions*. Computer J. 6 pp 308-320 1964

[7] Ledgard, H. F. *A formal system for defining the syntax and semantics of computer languag* MAC-TR-60 (Thesis), Project MAC, MIT April 1969.

[8] McCarthy, John, et al., *LISP1.5 Programmers Manual*. The M.I.T. Press, Cambridge, Mass., 1962.

[9] Newey, M. C. *Formal semantics of LISP with applications to program correctness.* Stanford Artificial Intelligence Laboratory, Memo AIM-257, January 1975 .

[10] Reynolds, J. C. Definitional interpreters for higher-order programming languages. *Proc. 25th National ACM Conference, Boston*, (1972).

[11] Scott, D. *Mathematical concepts in programming language semantics*. AFIPS VOL 40, AFIPS Press.

[12] Steele, Guy Lewis Jr. *LAMBDA: The Ultimate Declarative*. AI Memo 379. MIT AI Lab (Cambridge, November 1975).

[13] Steele, Guy Lewis Jr., Sussman, Gerald Jay. *LAMBDA: The Ultimate Imperative.* AI Memo 353. MIT AI Lab (Cambridge, March 1976).

[14] Steele, Guy Lewis Jr., Sussman, Gerald Jay. *The Revised Report on SCHEME A Dialect of LISP.* AI Memo 353. MIT AI Lab (Cambridge, January 1978).

[15] Fraser, A. G. *On the Meaning of Names in Programming Systems.* CACM Vol. 14, No. 6 pp 409-416 1971.

INDEX