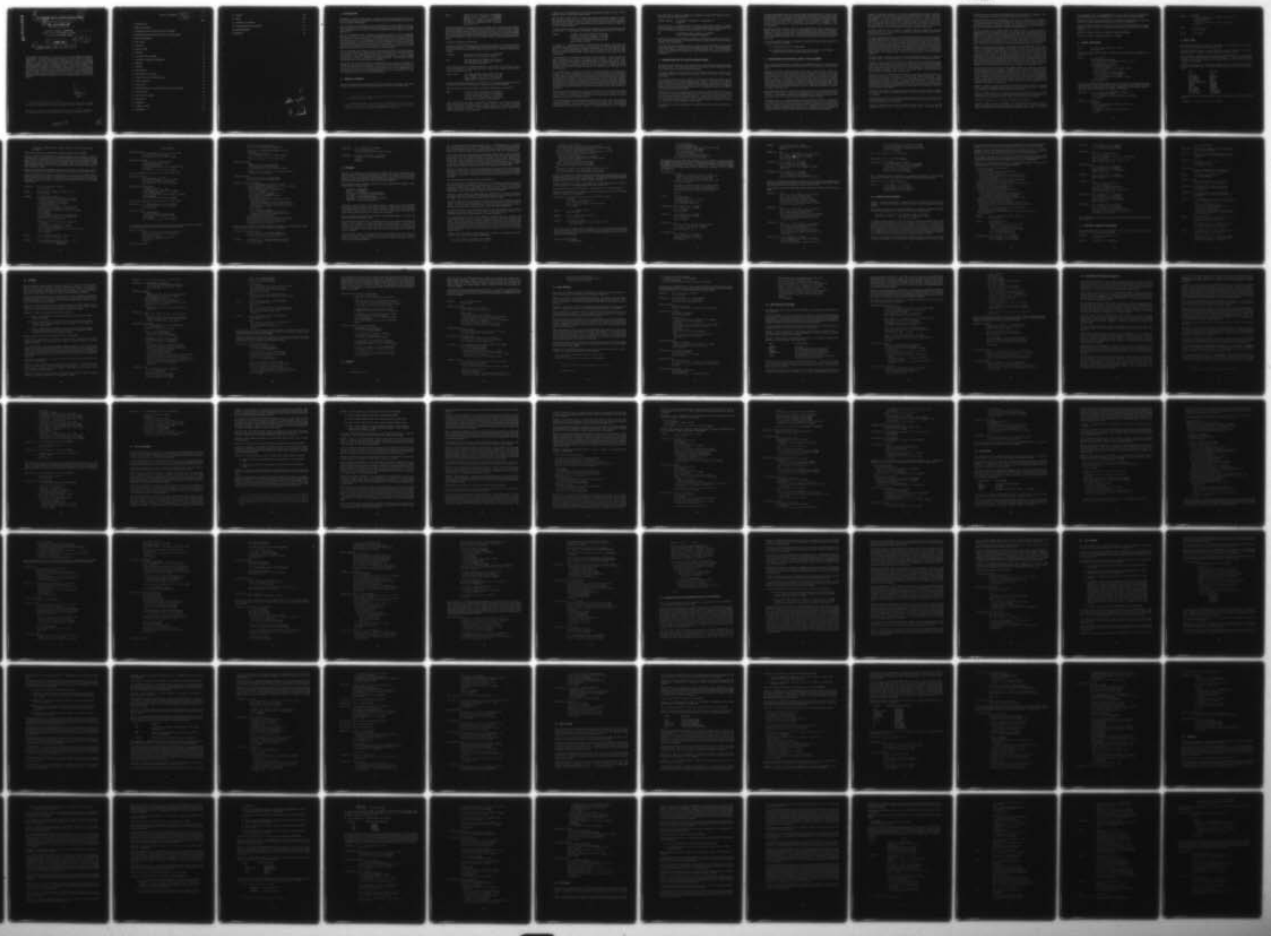
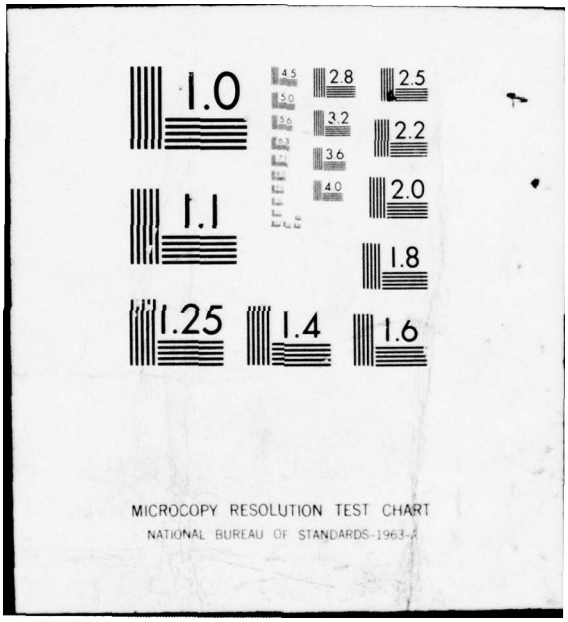
AD-A036 468 XEROX PALO ALTO RESEARCH CENTER CALIF COMPUTER SCIEN--ETC F/G 9/2
THE INTERLISP VIRTUAL MACHINE SPECIFICATION,(U)
SEP 76 J S MOORE N00014-75-C-0626

UNCLASSIFIED NL

1 OF 2
AD
A036468




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

THE INTERLISP VIRTUAL MACHINE SPECIFICATION

by J. Strother Moore, II*

COMPUTER SCIENCE LABORATORY
XEROX PALO ALTO RESEARCH CENTER
PALO ALTO, CALIFORNIA 94306

September 1976

131 p.

N00014-75-C-0626

The INTERLISP Virtual Machine is the environment in which the INTERLISP System is implemented. It includes such abstract objects as "Literal Atoms", "List Cells", "Integers", etc., the basic LISP functions for manipulating them, the underlying program control and variable binding mechanisms, the input/output facilities, and interrupt processing facilities. In order to implement the INTERLISP System (as described in *The INTERLISP Reference Manual* by W. Teitelman, et. al.) on some physical machine, it is only necessary to implement the INTERLISP Virtual Machine, since Virtual Machine compatible source code for the rest of the INTERLISP System can be obtained from publicly available files. This document specifies the behavior of the INTERLISP Virtual Machine from the implementor's point of view. That is, it is an attempt to make explicit those things which must be implemented to allow the INTERLISP System to run on some machine.

---

* Formerly at the Computer Science Laboratory, Xerox Palo Alto Research Center, Palo Alto, Ca. 94304, currently at the Computer Science Laboratory, Stanford Research Institute, Menlo Park, Ca. 94025.

This work was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-0626.

TABLE OF CONTENTS

1. INTRODUCTION

INTERLISP is an interactive LISP system. It consists of a large and sophisticated collection of user support facilities (such as DWIM and the Programmer's Assistant [TEI]) built on top of a fairly conventional LISP language.

We call this underlying conventional language "Virtual Machine" (or simply VM) LISP. The user support facilities are written entirely in VM LISP, and are in the public domain. Thus, if VM LISP is implemented on some machine, the rest of INTERLISP can be obtained from publicly available files[1].

Although the INTERLISP System is extensively documented at the user level in the INTERLISP Reference Manual [2], it is not possible to implement the system from that documentation. The purpose of this document is to specify VM LISP as fully as possible from the implementor's point of view. Consequently, this document emphasises clarity and conciseness over intuitive appeal. It is expected that a prospective implementor will have access to the INTERLISP Reference Manual for explanations of the justification or implications of certain specifications. Furthermore, since its purpose is mainly a practical one (i.e., to tell an implementor what must be done), the document is not altogether formal.

Because INTERLISP evolved under the rather sophisticated BBN TENEX[2] time sharing system, it assumes the presence of capabilities (such as user-defined interrupt characters) which may not be found in the implementor's environment. If an implementor is forced by such circumstances to forego the implementation of certain INTERLISP features, the user-support facilities may not perform as described in the Reference Manual. The implementor assumes responsibility for the documentation of such deficiencies.

A great deal of care has been taken in the preparation of this document to determine the assumptions made in the high-level facilities about features in the underlying VM. Because of the size and complexity of the system we cannot guarantee that we have identified them all, and therefore do not assure the prospective implementor that the rest of INTERLISP will run perfectly upon loading it into the just implemented VM. However, this document goes a long way toward that admirable (and probably impossible) goal.

2. PRIMITIVE CONCEPTS

Below we introduce several concepts and terms used throughout this document. We do not attempt formal definitions of these concepts because we feel they are sufficiently clear.

---

1 For information write Dr. W. Teitelman, Computer Science Laboratory, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, Ca. 94304.

2 The INTERLISP implementation on the DEC PDP-10, called INTERLISP-10, was developed under the management of researchers at Xerox Palo Alto Research Center and Bolt, Beranek and Newman. Comprehensive user documentation is provided in the INTERLISP Reference Manual [2].

object — Anything which can be given to an INTERLISP program as data or returned by an INTERLISP program as a result of a computation. Equivalently, an object is anything that can be the value of an INTERLISP variable. Examples: NIL, 143, (F X Y).

INTERLISP programs can dynamically create "new" objects using "creation functions" supplied by the Virtual Machine. These functions return objects that did not exist in the user's Virtual Machine immediately before the creation function was invoked. That is, they return objects that no other VM function could have returned prior to the invocation of the creation function. Most implementations accomplish the implied illusion of infinite space by secretly reclaiming the space occupied by an object once no VM LISP function can detect the absence of the object.

The details are presented in Section 28.

We will have occasion to talk about concepts which are not objects in the Virtual Machine but which have relevance to an implementation of the Virtual Machine. Such meta-objects include devices and buffers as well as mathematical entities such as sets, character sequences, and n-tuples.

meta-object — Any thing or concept, other than an INTERLISP object, which can be discussed in English.

form — Any object used as the argument to the function EVAL (cf. Section 16). Examples: NIL, 143, (F X Y) if they are given to EVAL.

Note that the determination of whether something is a form is made on the grounds of how it is used rather than how it is constructed or how it is written down. However, typically, forms are just List Structures or Atoms.

Value of a form — The object returned by EVAL when given the form. Examples: The value of NIL is NIL. The value of (ADD1 142) is 143. The value of (CONS (QUOTE F) (QUOTE (X Y))) is (F X Y).

Note that not all forms have values: some cause errors or otherwise alter the flow of control so that EVAL does not return to the point of invocation (e.g., the goto statement).

The next concept is probably the single most important concept used in this document.

field — A "place", usually associated with an object or meta-object, that can be used to "hold" another object or meta-object. Example: A List Cell is an object with two fields, named the CAR and CDR fields, each of which can hold an object.

There are two operations on fields: "accessing" and "replacing". If x is an object with a field which contains y, then it must be possible to compute y given x. This is called "accessing" the field. Furthermore, it must be possible to modify that field of x so that it is made to contain another object instead of y. This operation will be called either "replacing" (the contents of) the field or "setting" the field.

In general, only the implementor has full access and replacement rights on a field. In some instances the user is given limited rights to fields.

Note that fields are not objects: A variable may have as its value the contents of a given field, however a variable cannot have as its value the field itself. We will always be specific about whether the contents of a field is an object or meta-object, and what, if any, restrictions are placed on the contents. Unless otherwise stated, any field said to contain an object can contain any object whatsoever.

The final primitive concepts are concerned with communication between the VM and the "outside world". The most abstract and important of these concepts is that of the "character".

character — A graphic mark in the alphabet available to the machine's input/ouput facilities. Examples usually include such characters as 'A', 'a', and '(', as well as "non-printing" characters such as space, tab and form-feed.

A character is a meta-object because it exists outside the machine. We assume the implementor has designated a set of characters to be used in input/output transactions with the VM. This set will be called the "standard VM character set". For each character in this set there is a unique INTERLISP object either in the set of Literal Atoms or the set of Integers which is identified with that character. These particular objects are called Characters (note capitalization).

A certain subset of the characters are known as "control characters". These characters are usually "non-printing" (in the sense that outputting such a character causes no mark to be made) and usually perform control or formatting functions on certain physical devices. Since these characters are non-printing, we associate with each control character a printing character, called the "↑equivalent" (pronounced "uparrow equivalent"). Sometimes the control character will be printed by printing the character '↑' followed by the ↑equivalent of the control character.

It is assumed there is a character (and hence, a Character) called the "carriage return" character, which causes output devices to position their print mechanisms so that subsequent characters will be printed starting at the left-hand margin and immediately below the last line. In some systems, more than one character must actually be sent to certain devices to achieve this effect (e.g., one character to return the print mechanism to the left margin and another to advance the line). Reading and writing more than one character per carriage return character is permitted. In fact, the precise characters transferred may be device dependent. However, the implementor is expected to maintain the illusion of the single carriage return character by translating to and from the appopriate sequences when fetching and depositing characters (cf. Section 21).

We assume that there is a one to one mapping from the standard VM characters onto a subset of the integers. The INTERLISP Small Integers (cf. Section 9) in the range of the above mapping are called "character codes". The number of bits required to represent the largest character code is called "standard VM bytesize".

If the ASCII mapping is used, the integers are those from 0 to 127. The control characters are those with character codes from 0 to 31 and the character codes of their ↑equivalents are obtained by adding 64. Thus, the character "control-A" has character code 1, its ↑equivalent is the character 'A' and has code 65. "control-A" is sometimes printed as "↑A". If a mapping

other than ASCII is used, the implementor is expected to define these character sets in accordance with their properties above.

character sequence — a meta-object consisting of a succession of characters.

The ith character in a character sequence cannot be changed without producing a different character sequence. (In Section 12 we will introduce a meta-object, called a string, which allows its characters to be replaced without the production of a new meta-object.)

file — a meta-object which is used as a character source or sink for input/output operations.

Physically, files may be represented as sequences of character codes stored on a disc or other external storage device, or sequences of character codes coming from or going to any available input or output device.

Technically, files are meta-objects and not objects, because INTERLISP programs cannot directly manipulate them. However, INTERLISP assumes that each file is uniquely identified by a "file name" which is representable as an INTERLISP Literal Atom. The Virtual Machine provides for input/output on named files, using Literal Atoms to indicate the source or destination file.

3. CONVENTIONS FOR VM FUNCTION SPECIFICATIONS

This Section and the next explain the conventions used in this document when specifying the VM LISP functions. These conventions should not be confused with the INTERLISP facilities which allow the user to define new INTERLISP functions.

The precise nature of a VM LISP function is fully specified in Section 16. However some background information is necessary to understand the form and meaning of the function specifications.

In this document we use the word "function" in an extended mathematical sense to refer to the abstract association or mapping between some n-tuple of "arguments" and a value or effect. A function is named by an INTERLISP object called a Literal Atom (cf. Section 8) which contains a function object (cf. Section 16) in its function definition field. The function object is essentially a program, which tells EVAL how to compute the value and/or effect of the function named by the Literal Atom.

In this document, when we specify some function we will first write down the function name (a Literal Atom). Following that will be a list of the function's parameter names. Each name will be in lower case and separated from the others by ';'. The entire list will be enclosed in '[' and ']'. If the function takes an indefinite number of arguments we will use an ellipsis ("...") in the parameter list. (Such a function is called a "nospread" function; otherwise, the function is a "spread" function. See Section 16 for the details.)

The parameters are merely placeholders. The particular names used in this document are not important.

Following the parameter list we will write down English text which specifies, in terms of the parameter names given, the actions performed by the function when it is applied to some argument objects. As is made clear in the next Section, the parameter names are understood to represent the objects supplied as arguments. The text defining the behavior of the function, called the "body" of the specification, will be indented to distinguish it from surrounding explanatory material. There are numerous examples of such specifications in the following pages.

Sometimes (cf. AND in Section 5) we will write "(NOEVAL)" after the parameter list in a function specification. We say that the corresponding function object is "noeval-type". Otherwise, it is said to be "eval-type". Informally, whether a function object is eval-type or noeval-type determines whether EVAL will bind the parameter names to the values of the forms in the argument positions, or the forms themselves. (See the specification of EVAL in Section 18.)

From the implementor's viewpoint it is important to understand that each function specification in this document does two things:

(1) It specifies the nature of a function object.

(2) It specifies that the function object shall initially be found in the function definition field of a certain Literal Atom (the function name).

4. CONVENTIONS AND DEFINITIONS USED IN THIS DOCUMENT

We will use certain conventions and definitions when specifying functions. Usually they will be introduced before they are used the first time. Below we present those most commonly used.

Convention: Lower-case character sequences will be used as meta-variables to denote both INTERLISP objects and meta-objects. We will have occasion to refer both to the meta-variable itself and to the value (object or meta-object) it denotes. For example, we may wish to say "Let the meta-variable x denote the sum of the values currently denoted by the meta-variables x and y." To distinguish a meta-variable from its value we will use an underline. When a meta-variable is underlined the construction is understood to denote the value of the meta-variable. When not underlined the construction denotes the meta-variable itself. Thus, the above example can be abbreviated to "Let x be $\underline{x}+\underline{y}$." Note that if x denotes y (the meta-variable itself, not its denotation), then while "let x be 1" affects the denotation of x, "let $\underline{x}$ be 1" affects the denotation of y.

Note: The reader should not confuse meta-variables with the notion of variables provided by INTERLISP. Meta-variables are strictly a notational device for communicating with the reader. Variables (as implemented in INTERLISP) are INTERLISP objects, namely Literal Atoms, which are used as forms. This document carefully distinguishes the two concepts.

Convention: If $x_1, x_2, \ldots x_k$ denote objects and f denotes a VM function name, then whenever the specification of some computation uses the construct $f[\underline{x}_1;\underline{x}_2;\ldots\underline{x}_k]$ it is understood to imply that at that point in the computation the computation specified as defining the $\underline{n}$-ary function $\underline{f}$ should be executed with the successive $\underline{n}$ parameter names of $\underline{f}$ denoting the corresponding first $\underline{n}$ elements of the sequence $\underline{x}_1,\underline{x}_2,\ldots,\underline{x}_k$,NIL,NIL,NIL,..., and the construct is to denote the object (if any) "returned" (see the next convention) by that computation.

Convention: Successive sentences in the function specification body specify successive computational processes that are to be carried out sequentially when the function is applied to some arguments. We use clauses beginning with the words "if", "elseif", and "else" (separated by ";") to specify the conditional structure of the function. When the scope of a "then-clause" is ambiguous the entire clause is further indented. The phrase "return x" means that if a computation reaches that point of the specification then all subsequent statements in the specification are to be ignored (as specifying the computation along a different path through the function) and x is to be considered the value of the function application.

Convention: When we refer to objects in a boolean context (in constructions using the English words "if", "or", "and", and "not") the object NIL is identified with falsity and all other objects are identified with truth.

Convention: Whenever the body for some function specification does not specify an action for some possible argument combinations, the implementor is free perform any action desired. It is assumed this freedom will be used to merely avoid certain type checks (e.g., assume the argument to CAR is a List Cell and accept the consequences on other types of objects). Should the implementor's default action for any VM function be meaningful to the user (e.g., CAR of an atom always returning NIL) the implementor is expected to document the fact that such behavior is not standard. Furthermore, the implementor is expected to document those default actions which may result in harm to the user's Virtual Machine (e.g., a replacement function which, when improperly used, will destroy meaningful data or confuse the garbage collector).

Convention: When we refer to an object in the set of INTERLISP Integers we will capitalize the word "integer". We will leave it in lower case when referring to the mathematical entity.

Convention: We will write down integers and real numbers in standard mathematical notation in base-10. When referred to as objects they shall denote the corresponding INTERLISP Integer or Floating Point Number (cf. Section 10). In this document, all Floating Point Numbers will be written with at least one digit to the right of the decimal point to distinguish them from integers followed by periods. (That is, the real 10.0 will be written (in this document) with the redundant 0, to distinguish it from the integer 10.)

Convention: We will occasionally use meta-variables which denote INTERLISP Integers and Floating Point Numbers in constructions involving standard mathematical notation. In this context the meta-variables are understood to be abbreviations for the mathematical entities represented by their values. (That is, if x denotes an INTERLISP Integer -- an object which merely behaves somewhat like a certain mathematical entity -- then in the construction $\underline{x}+1$, x is treated as though it denotes the mathematical entity the Integer represents.) This convention allows the use of standard mathematical notation involving objects even though the notation is formally defined on meta-objects.

Convention: When we refer to a character sequence enclosed in quotation marks as though it were an object, it denotes an INTERLISP String (cf. Section 12) with the character sequence as its pname.

Convention: We will often use the name of a field to refer to the contents of the field, if such use is unambiguous. For example, we will refer to the CAR of a List Cell, when we mean the contents of the CAR field of the List Cell.

Convention: Whenever we say some computation should be done for each x in a specified sequence (e.g., "for i from 1 to $\underline{n}$ do ..." or "for each x_i do ...") we mean that the

computations should be performed in the same order as the x's occur in the sequence. That is, the computation for the first $\underline{x}$ should precede that for the second, etc.

We now present the commonly used definitions. The purpose of a definition is to introduce a suggestive phrase that has a precise formal meaning. Usually the defined phrase involves one or more meta-variables. Whenever an instance of a defined phrase is used the meaning is that obtained by reading the definition with the meta-variables of the definition denoting the objects or meta-objects indicated by the instance of the phrase used.

We will occasionally use an English variant of a defined phrase and expect the reader to recognize that we are still speaking formally. For example, later we define the phrase "the representation of x as an Integer". We may use the phrase "return the representation as an Integer of $\underline{x}$" or "represent $\underline{x}$ as an Integer and return it" or even "represent and return as an Integer $\underline{x}$." It is hoped that all three of these will obviously be understood by the reader to mean: "Let temp be the representation of $\underline{x}$ as an Integer. Return $\underline{temp}$." The reason we use such variants is that they occasionaly allow us to reduce the number of meta-variables the reader must contend with (as above) and they allow a more natural style of specification.

Definition: "f[x_1;...x_k]", where f denotes a non-VM function, means "APPLY*[$\underline{f}$;$\underline{x}_1$;...$\underline{x}_k$]". Since APPLY* is a VM function, this definition is meaningful. The reason we cannot appeal to the convention on VM function application (above) to make sense out of $\underline{f}$[$\underline{x}_1$;...$\underline{x}_k$] is that since $\underline{f}$ is not a VM function it does not have a specification in this document and the above convention on the meaning of VM function application was based on the body of the specification of the function. It also happens that while VM calls to other VM functions (as almost all calls in this document are) can be implemented by any technique desired, calls to user functions must use a well-defined stack structure defined in Sections 16-20. This definition makes this clear because APPLY* in fact manipulates the stack. Finally, for sanity, it should be pointed out that the effect and value of $\underline{f}$[$\underline{x}_1$;...$\underline{x}_k$] is in fact the same (except for the effect on the user's stack) as "APPLY*[$\underline{f}$;$\underline{x}_1$;...$\underline{x}_k$]" whether $\underline{f}$ is a VM function or not.

Definition: "cause error n with culprit x" means "ERRORX[LIST[$\underline{n}$;$\underline{x}$]]. Perform any unspecified (but presumably meaningful) computation". ERRORX is not in the VM but is defined as part of the user-support facilities of INTERLISP. It is the main entry into the error handling routines. Nominally ERRORX never returns to the computation which called it (i.e., to the compuation which "caused the error") but (using the stack manipulating functions discussed in Sections 17 and 18) returns to some higher process. However, the user can redefine ERRORX and therefore it may be altered so as to return control to the point of invocation. Implementations should therefore allow for this (by, for example, following the call to ERRORX by the equivalent of RETTO[T] (cf. Section 18)).

Definition: "pname of x" means "the character sequence that would be printed to a file other than the terminal by PRIN1[$\underline{x}$], when the radix field contains 10 (cf. Section 26). If PRIN1[$\underline{x}$] would cause error n with culprit z, then cause error $\underline{n}$ with culprit $\underline{z}$."

Definition: "PRIN2-pname of x with respect to y" means "the character sequence that would be printed to a file other than the terminal by PRIN2[$\underline{x}$;NIL;$\underline{y}$] when the radix field contains 10. If PRIN2[$\underline{x}$;NIL;$\underline{y}$] would cause error n with culprit z, then cause error $\underline{n}$ with culprit $\underline{z}$."

Definition: "the Literal Atom x" means "the Literal Atom whose pname is $\underline{x}$."

Convention: When we refer to a sequence of all capital characters as though it were an object, we mean the Literal Atom with that pname. Examples: NIL, T, LISTP. When such a sequence is underlined, it denotes the binding or value (in the EVALV sense -- see Section

18) of the Literal Atom. Thus, RANDSTATE means the Literal Atom with pname "RANDSTATE", while RANDSTATE means EVALV[RANDSTATE] -- the current value of that Literal Atom.

Definition: "the Character (note capitalization) corresponding to (the character) x", means "the Literal Atom or Integer whose pname consists only of the single character x."

Convention: When we refer to a character as a Character we mean the Literal Atom or Integer with the character as its pname. For example, we will refer to the ith Character in a character sequence.

Definition: A "Number" is either an Integer or a Floating Point Number.

Definition: An "Atom" is either a Literal Atom or a Number.

5. LOGICAL OPERATORS

```
EQ[x;y]          If x and y are the same object, return T;
                 else, return NIL.
```

The following function tests the equality of Numbers, and Stack Pointers (cf. Sections 9, 10, and 17).

```
EQP[x;y]         If x = y, return T;
                 elseif STACKP[x] and STACKP[y]:
                   If x and y contain the same frame extension, return T;
                   else, return NIL;
                 elseif NUMBERP[x] and NUMBERP[y]:
                   If FIXP[x] and FIXP[y]:
                     If x and y represent the same integer, return T;
                     else, return NIL;
                   else (x or y is a Floating Point Number):
                     If not FLOATP[x], let x be FLOAT[x].
                     If not FLOATP[y], let y be FLOAT[y].
                     If x and y represent the same real, return T;
                     else, return NIL.
                 else, return NIL.
```

Note: In a sense, EQP tests the equality of meta-objects contained in boxes (cf. Section 9). The implementor is free to extend EQP to test such equality on other classes of objects which use such representation (e.g., Strings). However, the next function, EQUAL, is responsible for the more general abstract equality of two objects.

```
EQUAL[x;y]       If x = y or EQP[x;y] or STREQUAL[x;y], return T;
                 elseif LISTP[x] and LISTP[y]:
                   return AND[EQUAL[CAR[x];CAR[y]];EQUAL[CDR[x];CDR[y]]];
                 else return NIL.

AND[x1;x2;...xk] (NOEVAL)
                 Let val be T.
                 For each xi (until some xi "evaluates to NIL") do:
                   Let val be EVAL[xi].
                   If val = NIL,
                     (we say xi "evaluated to NIL") return NIL.
                 Return val.
```

```
OR[x1;x2;...xk] (NOEVAL)
                For each xi (until some xi "evaluates to non-NIL") do:
                  Let val be EVAL[xi].
                  If val /= NIL,
                    (we say xi "evaluated to non-NIL") return val.
                Return NIL.

NOT[x]          If x=NIL, return T;
                else, return NIL.

NULL[x]         Return NOT[x]
```

6. DATA TYPES

Every object in the VM is a member of a unique class. All of the objects in a given class have certain common properties which define the class.

Associated with each class is a unique Literal Atom, called the "data type" of the class. Given any object it is possible to obtain the data type of the object's class.

The VM provides 11 primitive classes, plus facilities permitting the definition of new classes. Below we list the data types of the primitive VM classes. We will discuss the defining properties of each of these classes in the following Sections. Section 15 deals with the introduction of new classes.

Definition: A "data type" is a Literal Atom associated with a class of objects. No two classes may have the same data type. The initially existing classes and their data types are given below:

Class	Data Type
List Cells	LISTP
Literal Atoms	LITATOM
Small Integers	SMALLP
Large Integers	FIXP
Floating Point Numbers	FLOATP
Strings	STRINGP
Arrays	ARRAYP
Hash Arrays	HARRAYP
Stack Pointers	STACKP
Read Tables	READTABLEP
Terminal Tables	TERMTABLEP

The implementor may add additional primitive classes provided they are assigned unique data types.

```
TYPENAME[x]     Return the data type of the object x.
```

7. LIST CELLS

Definition: A "List Cell" (or "List Structure") is an object with two fields called the CAR field and the CDR field, each containing arbitrary objects. The List Cells constitute a distinct class of objects with class name LISTP.

The VM requires the existence of a field, called the "CONS count" field, which contains an integer. The initial contents of the CONS count field is 0 The functions which reference the CONS count field are CONS and CONSCOUNT.

```
LISTP[x]        If x is a List Cell, return x;
                else, return NIL.

CONS[x;y]       Increment the contents of the CONS count field by one
                and store the result in the CONS count field.
                Create and return a new List Cell
                with x in the CAR field and y in the CDR field.

CAR[x]          If LISTP[x],
                  return the contents of the CAR field of x;
                elseif LITATOM[x]:
                  If x is NIL, return NIL;
                  else, return any value desired (but cause no error).

CDR[x]          If LISTP[x],
                  return the contents of the CDR field of x;
                elseif LITATOM[x]:
                  If x is NIL, return NIL;
                  else, return any value desired (but cause no error).

RPLACA[cell;val]
                If cell=NIL:
                  If val=NIL, return NIL;
                  else, cause error 7 with culprit val;
                elseif LISTP[cell]:
                  Set the CAR field of cell to val.
                  Return cell;
                else, cause error 4 with culprit cell.

RPLACD[cell;val]
                If cell=NIL:
                  If val=NIL, return NIL;
                  else, cause error 7 with culprit val;
                elseif LISTP[cell]:
                  Set the CDR field of cell to val.
                  Return cell;
                else, cause error 4 with culprit cell.


LIST[x1;x2;...xk]
                Return CONS[x1;CONS[x2;...CONS[xk;NIL]...]]

CONSCOUNT[n]    If n is NIL, represent and return as an Integer
                (cf. Section 9) the integer contained in the CONS count
                field;
                else,
                  If not FIXP[n], let n be FIX[n].
                  Replace the contents of the CONS count field with
```

the integer represented by n and return n.

Definition: The "CDR chain from (some arbitrary object denoted by) x" is the ordered sequence of objects defined as follows: If x is not a List Cell, the CDR chain from x is the empty sequence. If x is a List Cell, the CDR chain from x is the sequence obtained by adding x to the front of the CDR chain from the CDR of x.

Note that CDR chains will be infinite if some List Cell occurs twice in the chain. If a computation is specified in terms of operations on the end of a CDR chain (e.g., involving the last List Cell in the CDR chain), the computation is considered to be unspecified for infinite chains.

Definition: A "proper list (of a sequence of n objects)" is the Literal Atom NIL if n is zero, and otherwise is a List Cell with the first object in the CAR field and a proper list of the remaining n-1 objects in the CDR field. The "length" of such a proper list is n. The "ith element" of a proper list of n objects, $1=<i=<n$, is the contents of the CAR field if i is 1, and otherwise is the i-1st element of the proper list in the CDR field. A "new" proper list is one for which new List Cells are in the CDR chain.

Note that these are definitions of terms we will use in this document. They do not define INTERLISP functions but merely allow us to refer to "proper lists" with precision. Also note that a proper list, x, always has a finite CDR chain. Furthermore, the CDR of the last List Cell in the CDR chain is always the Literal Atom NIL.

Convention: When we display a List Structure in this document we will use the notation produced by the function PRIN2. Thus (1 . 2) represents some List Cell with 1 in the CAR and 2 in the CDR, and (1 2 3) represents a proper list of the three Integers shown.

8. LITERAL ATOMS

Definition: A "Literal Atom is an object with the following properties:

(1) There is a field containing a (meta-object) character sequence called the "name" of the Literal Atom, such that no two distinct Literal Atoms have the same name and no Literal Atom has a name defined by <integer> or <floating point number> (cf. Sections 9 and 10). (It is permitted to limit the number of characters in the name of a Literal Atom. *The limit is unspecified*[3].)

(2) There is a field, called the "top-level value" field, which may contain any object.

(3) There is a field, called the "property list" field, which may contain any object.

---

3 INTERLISP-10 limits it to 99.

(4) There is a field, called the "function definition" field, which may contain any object.

The Literal Atoms consitute a distinct class of objects with class name LITATOM.

Informally, the name of a Literal Atom is the character sequence used to identify the object on input and output. The top-level value field contains the object to be interpreted as the top-level value of the Literal Atom when it is used as a variable in a form. The property list field usually contains a proper list and is used to associate additional information with the Literal Atom. When the Literal Atom is used as a function name (by being applied to some arguments), the contents of the function definition field is used as a program which should be run to compute the results.

The user has no access or replacement rights on the name field of a Literal Atom. However, the user can obtain the nth Character in the name of any Literal Atom (cf. NTHCHAR below).

Initially, the Literal Atom NIL shall exist and have NIL in its top-level value, property list, and function definition fields. In addition, the Literal Atom T shall exist and have T in its top-level value field. Of course, the names of all VM functions are also initially existing Literal Atoms with function objects (which behave according to the VM specifications) in their function definition fields.

LITATOM[x] — If x is a Literal Atom, return T;
else, return NIL.

ATOM[x] — If LITATOM[x] or FIXP[x] or FLOATP[x], return T;
else return NIL.

MKATOM[x] —
Let charseq be the pname of x.
If charseq conforms to the syntax of an Integer:
  represent and return as an Integer the integer
  denoted by charseq (cf. Section 9);
elseif charseq conforms to the syntax
of a Floating Point Number:
  represent and return as a Floating Point Number
  the real denoted by charseq (cf. Section 10);
elseif charseq is the name of a Literal Atom, litatom,
already created:
  return litatom;
elseif there are more characters in charseq than
the implementation allows in a Literal Atom name:
  cause error 11 with culprit NIL;
else:
  Create a new Literal Atom, litatom, whose name
  is charseq.
  Set the top-level value field of litatom
  to the Literal Atom NOBIND.
  Set the property list field and the function definition
  field of litatom to NIL.
  Return litatom.

PACK[x] — If x is a proper list of objects, $(x_1\ x_2\ \ldots\ x_k)$:
  return MKATOM[CONCAT[x_1;x_2;...x_k]];

PACKC[x] — If x is a proper list of objects, $(x_1\ x_2\ \ldots\ x_k)$:
  return MKATOM[CONCAT[CHARACTER[x_1];
                       CHARACTER[x_2];

...
CHARACTER[x_k]]];

GETTOPVAL[litatom]
If LITATOM[litatom], return the contents of the
top-level value field of litatom;
else, cause error 14 with culprit litatom.

SETTOPVAL[litatom;val]
If litatom is NIL and val is not NIL,
cause error 6 with culprit val;
elseif LITATOM[litatom]:
Set the top-level value field of litatom to val.
Return val;
else, cause error 14 with culprit litatom.

Note that SETTOPVAL maintains the top-level value of NIL at NIL.

GETPROPLIST[litatom]
If litatom is NIL, return NIL;
elseif LITATOM[litatom], return the contents
of the property list field of litatom;
else, cause error 14 with culprit litatom.

SETPROPLIST[litatom;proplist]
If litatom is NIL:
If proplist is NIL, return NIL;
else, cause error 7 with culprit proplist.
elseif LITATOM[litatom]:
Set the property list field of litatom to proplist.
Return proplist;
else, cause error 14 with culprit litatom.

Note that SETPROPLIST maintains the property list of NIL at NIL.

GETD[litatom] If LITATOM[litatom], return the contents of
the function definition field of litatom;
else, return NIL.

PUTD[litatom;defn]
If LITATOM[litatom]:
Replace the contents of the function
definition field of litatom with defn.
Return defn;
else, cause error 14 with culprit litatom.

The following three functions take Read Tables as arguments. These are objects that affect the way objects are printed. Read Tables are described in Section 22.

NCHARS[x;flg;rdtbl]
If flg, represent and return as an Integer the
number of characters in the PRIN2-pname of x
with respect to rdtbl;
else, represent and return as an Integer
the number of characters in the
pname of x.

NTHCHAR[x;n;flg;rdtbl]

```
                If not FIXP[n], let n be FIX[n].
                If  n<0, let n be NCHARS[x;flg;rdtbl]+n+1.

                If n<0 or n=0 or n>NCHARS[x;flg;rdtbl], return NIL;
                elseif flg:
                  return the nth Character
                  in the PRIN2-pname of x with respect to rdtbl;
                else:
                  return the nth Character in the pname of x.

UNPACK[x;flg;rdtbl]
                If flg:
                  Create and return a new proper list
                  containing the successive Characters
                  in the PRIN2-pname of x with respect to rdtbl;
                else:
                  Create and return a new proper list of the successive
                  Characters in the pname of x.

CHCON[x;flg;rdtbl]
                (Same specification as for UNPACK except
                use "character codes" for "Characters".)

DUNPACK[x;scratchlst;flg;rdtbl]
                If not LISTP[scratchlst], cause error 17 with
                culprit CONS["DUNPACK: SCRATCHLIST not a list";scratchlst];
                elseif scratchlst is a proper list:
                  If flg, let charseq be the PRIN2-pname of x
                  with respect to rdtbl;
                  else, let charseq be the pname of x.
                  Let n be the length of charseq.
                  If the length of scratchlst is greater
                  than or equal to n:
                    Let sublst be the terminal sublist of scratchlst
                    containing n elements.
                    Using RPLACA deposit the successive Characters in
                    charseq into the CAR fields of successive List Cells
                    in the CDR chain of sublst, starting with the first.
                    Return sublst;
                  else:
                    Return UNPACK[x;flg;rdtbl].
                    (Note:  The CARs of successive List Cells in
                    scratchlst may be replaced with Characters from
                    charseq before taking this exit.)
                else cause error 17 with culprit
                CONS["DUNPACK: unusual CDR in SCRATCHLIST";scratchlst].
                (Note:  The CARs of successive List Cells in
                scratchlst may be replaced with Characters from
                charseq before taking this exit.)
```

Note: The notes in DUNPACK allowing the CARs of scratchlist to be replaced before exiting permit the proper list check to be made as the function is running.

```
DCHCON[x;scratchlst;flg;rttbl]
                (Same specification as for DUNPACK except use
                CHCON for UNPACK and "DCHCON" for "DUNPACK" and
                use "character codes" for "Characters".)

CHCON1[x]       If the pname of x is the empty sequence, return NIL;
                else, return character code of the first character
                in the pname of x.
```

CHARACTER[n] If not FIXP[n], let n be FIX[n].
If n is a character code,
return the Character with character code n.

MAPATOMS[fn] For every Literal Atom, x, currently
represented in the Virtual Machine do:
fn[x].
Return NIL.

9. INTEGERS

Definition: The "Integers" (note capitalization) are objects that as far as possible obey the laws of arithmetic for integers (the mathematical entities). The Integers do not necessarily constitute a distinct class of objects. Some Integers must be so-called "Small" Integers (see below) with class name SMALLP. Unless all Integers are Small Integers, there must exist another class, with class name FIXP, containing the remaining Integers.

When characters are being read in (cf. Section 27) certain sequences denote Integers, namely those defined by <integer> below:

<oct digit> ::= 0|1|2|3|4|5|6|7
<digit> ::= <oct digit>|8|9
<oct seq> ::= <oct digit>Q|<oct digit><oct seq>
<oct integer> ::= <oct seq>|+<oct seq>|-<oct seq>
<dec seq> ::= <digit>|<digit><dec seq>
<dec integer> ::= <dec seq>|+<dec seq>|-<dec seq>
<integer> ::= <oct integer>|<dec integer>

A character sequence defined by <oct integer> denotes an Integer object which represents the positive or negative integer whose base-8 expansion is the sequence of octal digits given. A character sequence defined by <dec integer> denotes an Integer object which represents the positive or negative integer whose base-10 expansion is the sequence of decimal digits given. In both cases, if no sign (+ or -) is present, + is assumed.

The set of Integers is distinct from the subset of Floating Point Numbers (cf. Section 10) which have fractional part equal to zero.

The machine upon which the VM is implemented will have some internal representation of integers. This bit pattern is a meta-object, called an "unboxed value". It is usually not possible for the implementor to distiguish an arbitrary unboxed value from an address, and in particular, the address of some object. Therefore, Integers must usually be represented in some way other than by their unboxed values. There are two standard ways of representing Integers in the VM.

The first method exploits the knowledge that certain addresses, (e.g., those known to reference the machine instruction codes for the VM itself) cannot possibly point to objects. Any bit pattern which is such an address and is used as an object can then be treated as though it represented some Integer.

This representation has two desirable properties, noted in the definition below. Of course, only a relatively few Integers can be so represented, so it is desirable to represent the commonly used Integers in this fashion. Since the Integers occuring most frequently in user programs are clustered around 0, we call Integers represented in this fashion "Small Integers".

It is not usually the case that the bit pattern representing a Small Integer is also the unboxed value of the integer. Thus, the unboxed value of a Small Integer is obtained by applying some transformation to the bit pattern representing the Integer. This is called "unboxing" the Integer. The inverse transformation is applied to unboxed values to obtain a Small Integer. For example, if addresses less 2001 are to be considered Small Integers, and if it is desired to represent the integers -1000 to 1000 as Small Integers, then the unboxed value of a Small Integer would be obtained by subtracting 1000 from the address of the Small Integer.

Definition: A "Small" Integer is an Integer represented in such a way that two Small Integers represent the same integer if and only if the bit patterns representing the two Small Integers are identical. That is, no two distinct Small Integer objects represent the same meta-object. Consequently, Small Integers require little storage and boxing and unboxing them are efficient operations.

The VM requires that the character codes be Small Integers.

The second method of representing Integers is more general but consumes more space. Namely, the Integer is represented by the address of one or more storage locations known to contain the unboxed value of the Integer. The location is called a "box" and an Integer represented in such a way is called a "boxed" or "Large" Integer. Unboxing and boxing for boxed Integers is done by accessing and replacing the contents of the box.

Definition: A "Large" Integer is an Integer other than a Small Integer. The usual representation of a Large Integer is as a pointer to a storage location known to contain the unboxed value of the Integer. Two distinct Large Integers may represent the same integer.

In order to allow the user to discover how many boxes have been constructed, the VM requires the existence of a field, called the "Large Integer box count" field, which contains an integer. The initial contents of this field is 0. This field is updated during the process of constructing an Integer (see the definition below), and by the function BOXCOUNT.

The above discussion of boxes applies equally well to the implementation of Floating Point Numbers (see the next Section). In that case of course, an unboxed value is to be interpreted as the machine's representation of a Floating Point Number.

The Virtual Machine must allow for the possibility of arithmetic overflow or underflow. We assume the existence of a field, called the "arithmetic overflow flag" field, which contains either T, NIL, or (the Integer) 0. The initial contents of this field is 0. The contents can be changed with the function OVERFLOW (below) and determines the behavior of the VM in both Integer and Floating Point overflow and underflow. The definition below, which specifies the process of constructing the Integer representation of an integer, formally specifies the use of this field for Integer arithmetic (and a similar definition in Section 10 does so for Floating Point arithmetic).

Definition: The "representation of (the integer) x as an Integer", is the value of the meta-variable result (if any) after the following computation:

```
"If x is too large to be represented as an Integer:
   If the arithmetic overflow flag field contains T,
```

```
      cause error 5 with culprit 1;
    elseif the arithmetic overflow flag field contains NIL,
      let result be the representation of the largest possible Integer;
    else, let result be some unspecified Integer;
  elseif x is too small (large negative) to be represented as an Integer:
    If the arithmetic overflow flag field contains T,
      cause error 5 with culprit -1;
    elseif the arithmetic overflow field contains NIL,
      let result be the representation of the smallest (large
      negative) possible Integer;
    else, let result be some unspecified Integer;
  elseif x can be represented as a Small Integer,
    let result be the Small Integer representing x;
  else:
    Increment the contents of the Large Integer box count field by 1
    and store the result in the Large Integer box count field.
    Let result be a newly created boxed Integer representing x."
```

Note that if an overflow or underfow occurs while the arithmetic overflow flag field is 0, the Integer result of the above process is unspecified. The most natural behavior is that which would result if the overflow had not been detected: The Integer result represents whatever bit-pattern the hardware produced during the arithmetic operation.

Definition: The "floor of x", where x is a number, is the largest integer less than or equal to x. The "ceiling of x" is the smallest integer greater than or equal to x.

Thus, the floor of 2.7 is 2 and the ceiling is 3. The floor of -2.7 is -3 and the ceiling is -2.

Definition: The "integer part of x", where x is a number, is the floor of x, if x is non-negative, and is the ceiling of x, if x is negative.

```
OVERFLOW[flg]   Let oldflg be the contents of the arithmetic overflow
                flag field.
                If not flg = T and not flg = NIL,
                  let flg be the Integer 0.
                Set the arithmetic overflag flag field to flg.
                Return oldflg.


FIXP[x]         If x is an Integer, return x;
                else, return NIL.

SMALLP[x]       If x is a Small Integer, return x;
                else, return NIL.

IEQP[i;j]       If FIXP[i] and FIXP[j]:
                  If i and j represent the same integer, return T;
                  else, return NIL;
```

Note: IEQP is only specified for Integer arguments. This is so that the check can be made reasonably efficiently. That is, the two arguments can be unboxed and compared without regard for the consequences if they are in fact not Integers (provided the unboxing does not destroy the state of the VM).

```
SETN[nvar;valform] (NOEVAL)
                If LITATOM[nvar]:
```

```
                Let n be EVAL[nvar].
                Let val be EVAL[valform].
                If not NUMBERP[val], cause error 10 with culprit val;
                elseif n is neither a boxed Integer
                nor a boxed Floating Point Number,
                  return SET[nvar;val];
                else, store the unboxed value of val
                in the box associated with n, and return n;
              else, cause error 14 with culprit nvar.
```

Note that if the box itself affects the determination of what number its contents represents, then SETN[nvar;valform] will not necessarily make nvar represent the same number as valform. For example, if i is a boxed Integer and z is a Floating Point Number, then SETN[i;z] merely deposits the unboxed value of z into the box associated with i. When i is used, the contents of that box will be interpreted as an integer. That integer will usually not be the number represented by z.

```
BOXCOUNT[type;n]
                If n=NIL:
                  If type=NIL, represent and return as an Integer the
                  integer in the Large Integer box count field;
                  else, represent and return as an Integer the
                  integer in the Floating Point Number box count field.
                else, let n be FIX[n].

                If type=NIL, replace the contents of the Large Integer
                box count field with the integer represented by n;
                else, replace the contents of the Floating Point Number
                box count field with the integer represented by n.

                Return n.

FIX[n]          If FIXP[n], return n;
                elseif FLOATP[n]:
                  Represent and return as an Integer
                  the integer part of n;
                else, FIX[ERRORX[LIST[10;n]]].

IGREATERP[i;j]  If not FIXP[i], let i be FIX[i].
                If not FIXP[j], let j be FIX[j].
                If i > j, return T;
                else, return NIL.

ILESSP[i;j]     If not FIXP[i], let i be FIX[i].
                If not FIXP[j], let j be FIX[j].
                If i < j, return T;
                else, return NIL.

IPLUS[n1;n2;...nk]
                For each ni, if not FIXP[ni], let ni be FIX[ni].
                If k is zero, return the Small Integer 0;
                else, represent and return as an Integer
                the integer n1+n2+...+nk.

IDIFFERENCE[i;j]
                If not FIXP[i], let i be FIX[i].
                If not FIXP[j], let j be FIX[j].
                Represent and return as an Integer
                the integer i-j.
```

IMINUS[n] If not FIXP[n], let n be FIX[n].
Represent and return as an Integer the
integer -n.

ITIMES[n_1;n_2;...n_k]
For each n_i, if not FIXP[n_i], let n_i be FIX[n_i].
If k is zero, return the Small Integer 1;
else, represent and return as an Integer
the integer n_1*n_2*...n_k.

IQUOTIENT[i;j] If not FIXP[i], let i be FIX[i].
If not FIXP[j], let j be FIX[j].
If j=0, cause error 5 with culprit j.
Represent and return as an Integer the integer
part of i/j.

IREMAINDER[i;j] If not FIXP[i], let i be FIX[i].
If not FIXP[j], let j be FIX[j].
If j=0, cause error 5 with culprit j.
Return IDIFFERENCE[i;ITIMES[IQUOTIENT[i;j];j]].

Definition: The "N-bit binary expansion of (Integer) n" is the ordinary binary representation of the integer (represented by) n, in either 1 or 2's complement notation (implementor's choice) and employing N bits, with the high-order bits (and sign) to the left.

In the following, N must be at least large enough to allow an N-bit binary expansion of every Integer.

LOGAND[n_1;n_2;...n_k]
For each n_i, if not FIXP[n_i], let n_i be FIX[n_i].
If k is zero, return an Integer whose
N-bit binary expansion contains all 1's;
else, return an Integer whose N-bit binary
expansion has a 1 in bit position j ($1 \leq j \leq N$),
if and only if the N-bit binary expansion of each
n_i has a 1 in bit position j.

LOGOR[n_1;n_2;...n_k]
For each n_i, if not FIXP[n_i], let n_i be FIX[n_i].
If k is zero, return an Integer whose
N-bit binary expansion contains all 0's;
else, return an Integer whose N-bit binary expansion
has a 1 in bit position j ($1 \leq j \leq N$), if and only
if the N-bit binary expansion of some n_i has a
1 in bit position j.

LOGXOR[n_1;n_2;...n_k]
For each n_i, if not FIXP[n_i], let n_i be FIX[n_i].
If k is zero, return an Integer whose N-bit
binary expansion contains all 0's;
else, return an Integer whose N-bit binary
expansion has a 1 in bit position j ($1 \leq j \leq N$),
if and only if an odd number of the n_i have 1's in
bit position j.

LLSH[n;factor] If not FIXP[n], let n be FIX[n].
If not FIXP[factor], let factor be FIX[factor].
Return an Integer whose
N-bit binary expansion is obtained from that of n

by shifting it factor bit positions to the left (and filling with 0's) if factor>0, and shifting if factor bit positions to the right (and filling with 0's) if factor<0.

Note: "LLSH" stands for "logical left shift".

LRSH[n;factor] Return LLSH[n;IMINUS[factor]].

LSH[n;factor] If not FIXP[n], let n be FIX[n].
If not FIXP[factor], let factor be FIX[factor].
If the floor of n*2↑factor can be represented as an Integer, represent and return as an Integer the floor of n*2↑factor;
else, return an unspecified Integer.

Note: In INTERLISP-10 LSH is implemented as an arithmetic shift instruction. If the high-order bits are lost on the shift, the result is just the Integer representing the remaining bits.

RSH[n;factor] Return LSH[n;IMINUS[factor]].

GCD[i;j] If not FIXP[i], let i be FIX[i].
If not FIXP[j], let j be FIX[j].
Represent and return as an Integer the greatest common divisor of i and j.

10. FLOATING POINT NUMBERS

Definition: "Floating Point Numbers" are objects that as far as possible obey the laws of real arithmetic. The Floating Point Numbers constitute a distinct class of objects with class name FLOATP.

During input (cf. Section 27), Floating Point Numbers are denoted by character sequences defined by <floating point number> given below in terms of the Integer syntax:

```
<dec real> ::= <dec integer>.<dec seq>|<dec integer>.|.<dec seq>
<floating point number> ::= <dec real>|<dec integer>E<dec integer>|
                            <dec real>E<dec integer>
```

A character sequence defined by <dec real> denotes a Floating Point Number object which represents the real number whose decimal expansion is the sequence of characters given, followed by an infinite sequence of 0's. In the absence of a sign (+ or -), + is assumed. A sequence defined by <dec integer>E<dec integer> denotes a Floating Point Number object which represents the real obtained by multiplying the first denoted integer by 10 raised to the power denoted by the second (e.g. 125E3 denotes a Floating Point Number representing the real 125000.0.) A sequence defined by <dec real>E<dec integer> is interpreted analogously (e.g., 125.4E3 denotes a Floating Point Number representing the real 125400.0).

Although a given Floating Point Number represents exactly one real, it is not the case that any real can be represented. It is recognized that Floating Point Numbers inherently have a finite

magnitude and precision. Neither the maximum magnitude nor the minimum precision is specified since these quantities are largely determined by the host machine's architecture.

Definition: We say "(the Floating Point Number) x represents (the real) y to maximum precision" when the real deviation between y and the real denoted by x is as small as possible given the host machine's internal representation of Floating Point Numbers.

The VM requires the existence of a field, called the "Floating Point Number box count" field, which contains an integer. The initial contents of the field is 0. The field is updated by the process which constructs Floating Point Numbers and by the function BOXCOUNT.

Definition: The "representation of (the real) x as a Floating Point Number" is the value of the meta-variable result (if any) after the following computation:

```
"If x is too large to be represented as a
Floating Point Number:
  If the arithmetic overflow flag field contains T,
    cause error 5 with culprit 1.0;
  elseif the arithmetic overflow flag field contains NIL,
    let result be the representation of the largest possible
    Floating Point Number;
  else, let result be some unspecified Floating Point Number.
elseif x is too close to 0 to be represented
as a Floating Point Number, let result be the representation as a
Floating Point Number of the real 0.0;
elseif x is too small (large negative) to be
represented as a Floating Point Number:
  If the arithmetic overflow flag field contains T,
    cause error 5 with culprit -1.0;
  elseif the arithmetic overflow flag field contains NIL,
    let result be the representation of the smallest
    (large negative) possible Floating Point Number;
  else, let result be some unspecified Floating Point Number.
elseif x is to be represented as a boxed
Floating Point Number (implementor's choice):
  Increment the contents of the Floating Point Number box count
  field by 1 and store the result in the Floating Point Number box
  count field.
  Let result be a newly created boxed Floating Point Number
  representing x to maximum precision;
else, let result be the unboxed Floating Point Number representing x to
maximum precision."

FLOATP[x]          If x is a Floating Point Number, return x;
                   else, return NIL.

FLOAT[n]           If FLOATP[n], return n;
                   elseif FIXP[n]:
                     Represent and return as a Floating Point
                     Number the real obtained by
                     appending a decimal point followed by
                     an infinite sequence of 0's to the right
                     of the decimal expansion of n.
                   else, FLOAT[ERRORX[LIST[10;n]]].

FGREATERP[x;y]     If not FLOATP[x], let x be FLOAT[x].
                   If not FLOATP[y], let y be FLOAT[y].
                   If x > y, return T; else, return NIL.
```

FLESSP[x;y] If not FLOATP[x], let x be FLOAT[x]..
If not FLOATP[y], let y be FLOAT[y].
If x < y, return T; else return NIL.

FPLUS[n_1;n_2;...n_k]
For each n_i, if not FLOATP[n_i],
let n_i be FLOAT[n_i].
If k is zero, represent and return as a Floating Point
Number the real 0.0;
else, represent and return as a Floating Point
Number the real $n_1+n_2+\ldots+n_k$.

FDIFFERENCE[x;y]
If not FLOATP[x], let x be FLOAT[x].
If not FLOATP[y], let y be FLOAT[y].
Represent and return as a Floating Point
Number the real x-y.

FMINUS[n] If not FLOATP[n], let n be FLOAT[n].
Represent and return as a Floating Point
Number the real -n.

FTIMES[n_1;n_2;...n_k]
For each n_i, if not FLOATP[n_i],
let n_i be FLOAT[n_i].
If k is zero, represent and return as a
Floating Point Number the real 1.0;
else, represent and return as a Floating Point
Number the real $n_1*n_2*\ldots*n_k$.

FQUOTIENT[i;j] If not FLOATP[i], let i be FLOAT[i].
If not FLOATP[j], let j be FLOAT[j].
If j=0.0, cause error 5 with culprit j.
Represent and return as a Floating Point
Number the real i/j.

FREMAINDER[x;y] If not FLOATP[x], let x be FLOAT[x].
If not FLOATP[y], let y be FLOAT[y].
If y = 0.0, cause error 5 with culprit y.
Represent and return as a Floating Point Number
the real representing the difference between x and
the unboxed value of (x/y)*y.

Note: FREMAINDER is non-zero only due to the finite precision of the host machine's floating point arithmetic.

11. ADDITIONAL ARITHMETIC FUNCTIONS

The following VM functions could be defined in terms of those in the last two Sections. However, it is useful to consider them primitive.

NUMBERP[x] If FIXP[x] or FLOATP[x], return T;
else return NIL.

MINUSP[x] If FLOATP[x], return FMINUSP[x];

else return IMINUSP[x].

GREATERP[x;y] If FLOATP[x] or FLOATP[y], return FGREATERP[x;y];
else return IGREATERP[x;y].

LESSP[x;y] (Same specification as for GREATERP except use FLESSP instead of FGREATERP and ILESSP instead of IGREATERP.)

PLUS[x_1;x_2;...x_k]
If FLOATP[x_i], for any $1=<i=<k$:
FPLUS[x_1;x_2;...x_k];
else IPLUS[x_1;x_2;...x_k].

DIFFERENCE[x;y]
(Same as GREATERP except use FDIFFERENCE for FGREATERP and IDIFFERENCE for IGREATERP.)

MINUS[x] If FLOATP[x], return FMINUS[x];
else return IMINUS[x].

TIMES[x_1;x_2;...x_k]
(Same specification as for PLUS except use FTIMES for FPLUS and ITIMES for IPLUS.)

QUOTIENT[x;y] (Same specification as for GREATERP except use FQUOTIENT for FGREATERP and IQUOTIENT for IGREATERP.)

REMAINDER[x;y] (Same specification as for GREATERP except use FREMAINDER for FGREATERP and IREMAINDER for IGREATERP.)

EXPT[x;y] If not NUMBERP[x], cause error 10 with culprit x;
elseif not NUMBERP[y], cause error 10 with culprit y.

If FIXP[x] and FIXP[y] and y>=0:
Represent and return as an Integer x↑y;
elseif x<0 and not EQP[y;FIX[y]]:
Cause error 17 with culprit
CONS["Illegal exponentiation:"LISI[:EXPT:x:y]];
else, represent and return as a Floating Point Number the real x↑y.

SQRT[x] If not NUMBERP[x], cause error 10 with culprit x;
elseif x<0, cause error 17 with culprit
CONS["SQRT of negative value":x].

Represent and return as a Floating Point Number the square root of x (note that x may be a Floating Point Number or an Integer).

LOG[x] If not NUMBERP[x], cause error 10 with culprit x;
elseif x<0, cause error 17 with culprit
CONS["LOG of negative value":x].

Represent and return as a Floating Point Number the natural logarithm of x (note that x may be a Floating Point Number or an Integer).

ANTILOG[x] If not NUMBERP[x], cause error 10 with culprit x.

Represent and return as a Floating Point Number the real whose natural logarithm is x (note that x may be a Floating Point Number or an Integer).

SIN[x;radiansflg]

If not NUMBERP[x], cause error 10 with culprit x.

Represent and return as a Floating Point Number the sine of x (measured in radians if radianflg, otherwise in degrees) (Note that x may be a Floating Point Number or an Integer.)

COS[x;radiansflg]

(Same specification as for SIN except use "cosine" instead of "sine".)

TAN[x;radiansflg]

(Same specification for as SIN except use "tangent" instead of "sine".)

ARCSIN[x;radiansflg]

If not NUMBERP[x], cause error 10 with culprit x; elseif x < -1 or x > 1, cause error 17 with culprit CONS["ARCSIN: arg not in range";x].

Represent and return as a Floating Point Number the angle (measured in radians if radianflg and otherwise measured in degrees) between -90 and +90 degrees whose sine is x (note that x may be a Floating Point Number or an Integer).

ARCCOS[x;radianflg]

(Same specification as for ARCSIN except use "cosine" for "sine", "ARCCOS" for "ARCSIN" "0" for "-90" and "180" for "+90".)

ARCTAN[x;radianflg]

If not NUMBERP[x], cause error 10 with culprit x.

Represent and return as a Floating Point Number the angle (measured in radians if radianflg and otherwise measured in degrees) between 0 and 180 degrees whose tangent is x (note that x may be a Floating Point Number or an Integer).

The following function, RAND, is used to generate pseudo-random numbers. It is assumed that in order to so operate, RAND must save some state information from one call to the next. The VM assumes this state information (called a "RAND state") is contained in an implementor defined object (called a "RAND State" - note capitalization) and is stored in the value field of the Literal Atom RANDSTATE. The VM also assumes that a RAND State can be destructively modified so as to represent any given RAND state. (Thus, a RAND State might be a boxed

Integer capable of representing many integers. In INTERLISP-10 it is a List Cell containing two boxed Integers.) This allows RAND to save its new (next) state in the object representing its current state, thereby avoiding the creation of a new object. Finally, it is assumed that such a state entirely determines the next number generated by RAND (for a given pair of arguments). (That is, if a copy of the current RAND State is saved and then RAND is used to generate some sequence of "random" numbers, the same sequence can be generated in the future by restoring the saved State (with RANDSET) and executing the same sequence of calls to RAND.)

```
RAND[lower:upper]
                Let stateobj be RANDSTATE.
                If stateobj is not a RAND State:
                  Let stateobj be RANDSET[T].

                If FIXP[lower] and FIXP[upper]:
                  Using the RAND state in stateobj as
                  the current state, generate a psuedo-random integer,
                  i, lower=<i=<upper, and a new state, s.
                  Destructively modify stateobj so that it represents s.
                  Represent and return as an Integer the integer i.
                else:
                  Let lower be FLOATP[lower].
                  Let upper be FLOATP[upper].
                  Using the RAND state in stateobj as the current
                  state, generate a psuedo-random real,
                  x, lower=<x=<upper, and a new state, s.
                  Destructively modify stateobj so that it represents s.
                  Represent and return as a Floating Point Number
                  the real x.

RANDSET[state]  If state=T:
                  Let newstate be a new RAND state created from any
                  (pseudo-) random source available (e.g., a run-time
                  clock).
                  Let newstateobj be a new RAND State (note upper case)
                  representing newstate (i.e., newstateobj is a new object
                  as well as being a representation of a new state).
                  SETQ[RANDSTATE;newstateobj].
                elseif state is an object that represents a
                valid RAND state:
                  Let newstate be the RAND state represented
                  by state.
                  Let newstateobj be a new RAND State (note upper case)
                  representing newstate (i.e., newstateobj is a new object
                  representing the old state represented by state).
                  SETQ[RANDSTATE;newstateobj].
                elseif state /= NIL:
                  Cause error 17 with culprit
                  CONS["arg not previous value of RANDSET";state].

                Return a new RAND State (note upper case)
                representing the state represented by RANDSTATE
                (i.e., return a new object which
                represents the current RAND state).
```

12. STRINGS

Strings are objects which represent character sequences. However, the String handling functions expose a certain amount of the internal represention of Strings. It is possible to form two distinct Strings which share the same internal structure. This can be detected by replacing the characters in either String and observing side-effects on the other. Therefore, Strings have a richer and more complicated structure than mere character sequences.

We must first introduce the concept of a "string" (note lower case). Intuitively, a string is like a character sequence in that it specifies some succession of characters. However, unlike a character sequence, the ith character in a string can be changed without producing a new string. This can be formalized as follows:

Definition: A "string (of length n)" is a meta-object having n fields, each identified with an integer, $1 \leq i \leq n$ (provided $n > 0$), and each containing a Character. At any instant a string "represents" the character sequence with the same succession of Characters.

Definition: A "String" is an object with the following properties:

(1) There is a field, called the "source" field, which may contain either a character sequence or a string.

(2) There is a field, called the "position" field, which contains a positive integer, with the restriction that the integer cannot be greater than the number of characters in the source.

(3) There is a field, called the "charcount", which contains a non-negative integer, with the restriction that the position plus charcount of a String cannot exceed the number of characters in the source of the String.

Strings constitute a distinct class of objects with class name STRINGP.

At any instant, a given String, x, with position, i, and charcount, n, represents the character sequence consisting of the n characters in the source of x, starting at the ith. This is the pname of the String.

The reason the source field may contain either a character sequence or a string is that it is convenient to produce Strings directly from the contents of the name fields of Literal Atoms without converting those character sequences into character strings (cf. MKSTRING). Of course, since it is impossible to change the characters in a character sequence, the source of such a String must be replaced by a string the first time a character is to be changed (cf. *RPLSTRING*).

Definition: An "empty String" is one with charcount 0. The source and position fields of an empty String are irrelevant.

Definition: "Create a new String representing (character sequence) x" means "Create a new String, with source set to a new string representing x, position set to 1, and charcount set to the length of x." Whenever the source, position, or charcount of a new String is not as specified above, we will be explicit.

NCHARS (cf. Section 8) returns the number of characters in the pname of a String. NTHCHAR (cf. Section 8) returns the ith Character in the pname of a String.

STRINGP[x] If x is a String, return x; else return NIL.

STREQUAL[x;y] If STRINGP[x] and STRINGP[y]:
If the character sequence represented by x is the same as that represented by y, return x; else, return NIL.

MKSTRING[x;flg;rdtbl]
If flg:
Create and return a new String representing the PRIN2-pname of x with respect to rdtbl.
elseif STRINGP[x], return x;
elseif LITATOM[x], create and return a new String with source set to the name of x, position set to 1, and charcount set to the number of characters in the name of x;
else:
Create and return a new String representing the pname of x.

CONCAT[x_1;x_2;...x_n]
If n is zero, create and return a new empty String;
else:
Create and return a new String representing the character sequence obtained by concatenating the pnames of x_1 through x_n (in that order).

RPLSTRING[str;n;newchars]
If not STRINGP[str], let str be MKSTRING[str].
If n is NIL, let n be 1;
elseif not FIXP[n], let n be FIX[n].

If n<0, let n be NCHARS[str]+n+1.
If n<0 or n=0 or n>NCHARS[str],
cause error 27 with culprit newchars.

If newchars is a Literal Atom or a String
and n+NCHARS[newchars]-1 > NCHARS[str],
cause error 27 with culprit newchars;
else:
If the source of str is a character sequence (rather than a string), replace the source of str with a string representing the source of str.
Let strsource be the source of str.
Let i be the position of str.
Let l be the charcount of str.
Replace the contents of the successive Character fields of strsource, starting with the i+n-1st, with the successive Characters from the pname of newchars (from left-most through right-most), and if this process requires the replacement of a field beyond the i+l-1st one, cause error 27 with culprit newchars.
Return str.

SUBSTRING[str;n;m]
If not STRINGP[str] and not LITATOM[str],
let str be MKSTRING[str].
If n is NIL, let n be 1;
elseif not FIXP[n], let n be FIX[n].
If m is NIL, let m be NCHARS[str];
elseif not FIXP[m], let m be FIX[m].

```
                If n<0, let n be NCHARS[str]+n+1.
                If m<0, let m be NCHARS[str]+n+1.
                If n<0 or n=0 or n>m, return NIL.

                If STRINGP[str]:
                  Let i be the contents of the position field of str.
                  Create and return a new String with source
                  set to the source of str, position to i+n-1,
                  and charcount to m-n+1;
                else:
                  Create and return a new String with source
                  set to the name of str, position to n, and charcount
                  to m-n+1.

GNC[str]        If not STRINGP[str], let str be MKSTRING[str].
                If str is an empty string,
                  return NIL;
                else:
                  Let y be the first Character in the pname of str.
                  Let i be the contents of the position field of str.
                  Set the position field of str to i+1.
                  Return y.

GLC[str]        If not STRINGP[str], let str be MKSTRING[str].
                If str is an empty string,
                  return NIL;
                else:
                  Let y be the last Character in the pname of str.
                  Let n be the charcount of str.
                  Set the charcount field of str to n-1.
                  Return y.
```

The next function searches one string for the first occurrence of another. However, wild card characters are allowed. Thus, we will define the notion of two character sequences being equal with respect to some wild card character:

Definition: "(character sequences) seq_1 and seq_2 (each of length n) are equal with respect to the wild card skip", where skip is an arbitrary object, means "For each i from 1 to n, either the ith Character in seq_1 is skip or is the ith Character in seq_2."

```
STRPOS[pat;str;start;skip;anchor;tail]
                If start=NIL, let start be 1;
                elseif NUMBERP[start] and start<0:
                  Let start be NCHARS[str]+start-1.

                If not FIXP[start], let start be FIX[start].
                Let patlen be the length of the pname of pat.
                Let strlen be the length of the pname of str.
                If anchor, let max be start;
                else, let max be strlen-patlen+1.

                If there is an integer, i, start=<i=<max,
                such that the pname of pat and the patlen long
                substring of the pname of str starting at i are
                equal with respect to the wild card skip:
                  Let i be the smallest value denoted by such an i.
                  If tail, represent and return as an Integer i+patlen+1;
                  else, represent and return as an Integer i;
                else, return NIL.
```

The following two functions are used to search strings for the first occurrence of any one of a set of characters. In order to make this efficient, the VM allows the user to call a function (MAKEBITTABLE) which preprocesses a proper list of characters codes and produces an object which represents the corresponding set of characters in an efficient way. We call this representation of a set of characters a "bittable". The implementor is free to represent bittables in any way desired[4]. The VM allows for the possibility that the object representing a bittable can be modified (by replacing the contents of fields within it) so that the set it represents is changed.

```
MAKEBITTABLE[lst:complimentflg;oldbittable]
                If lst is not a proper list,
                  Let charset be an unspecified set of characters;
                else:
                  Let charset be the set of precisely those characters, c,
                  such that either (1) the character code of c is in
                  (the proper list) lst, or (2) c is the first character
                  in the pname of some non-FIXP element of lst.

                If complimentflg, let charset be the compliment
                of charset with respect to the set of all characters.

                If oldbittable is an object representing a bittable
                and can be modified to represent charset:
                  Modify oldbittable so that it represents charset.
                  Return oldbittable;
                else, create and return a new bittable representing
                charset.

STRPOSL[bittable:str;start:complimentflg]
                If start=NIL, let start be 1;
                elseif NUMBERP[start] and start<0:
                  Let start be NCHARS[str]+start-1.

                If not FIXP[start], let start be FIX[start].
                If bittable is not a bittable,
                  let bittable be MAKEBITTABLE[bittable].

                Let charset be the set represented by bittable.
                If complimentflg, let charset be the compliment
                of charset with respect to the set of all characters.

                If there is an integer, i, start=<i=<NCHARS[str],
                such that the ith character in the pname of str
                is in charset:
                  Let i be the smallest value denoted by such an i.
                  Represent and return as an Integer the integer i;
                else, return NIL.
```

13. ARRAYS

---

4 In INTERLISP-10 they are arrays.

Definition: "Arrays" are objects that contain a fixed number of fields, each identified by a positive Integer "subscript". An Array containing n fields is said to have size n. There are two kinds of Arrays which differ according to the class of objects which may be contained in their fields. The fields in a "Pointer" Array may contain any objects whatsoever. The fields in an "Integer" Array may only contain Integers. Arrays constitute a distinct class of objects with class name ARRAYP.

It is assumed that the implementation will take advantage of the restriction on the contents of Integer Arrays to avoid unnecessary boxing and unboxing of Integers. The implementor is free to provide arrays of other types and to generalize the functions ARRAY, ARRAYTYP, ELT, and SETA to handle them.

```
ARRAYP[x]       If x is an Array, return x
                else return NIL.

ARRAY[n;typ;initval]
                If not FIXP[n], let n be FIX[n].
                If n<0, cause error 27 with culprit n.

                If typ = FIXP:
                  If initval is NIL, let initval be 0;
                  elseif not FIXP[initval], let initval be FIX[initval].
                  Create and return a new Integer Array of size n,
                  each field of which initially contains (unboxed) initval;
                elseif typ = NIL or typ = POINTER:
                  Create and return a new Pointer Array of size n,
                  each field of which initially contains initval.

ARRAYSIZE[array]
                If ARRAYP[array],
                  return an Integer representing the the size of array
                else, cause error 28 with culprit array.

ARRAYTYP[array] If ARRAYP[array]:
                  If array is an Integer Array, return FIXP;
                  elseif array is a Pointer Array, return POINTER;
                else, cause error 28 with culprit array.

ELT[array;n]    If not ARRAYP[array], cause error 28 with culprit array.
                If not FIXP[n], let n be FIX[n].
                If 1=<n and n=<ARRAYSIZE[array]:
                  If array is an Integer Array,
                    represent and return as an Integer the integer
                    represented by the contents of the nth field of array;
                  elseif array is a Pointer Array,
                    return the contents of the nth field of array;

SETA[array;n;val]
                If not ARRAYP[array], cause error 28 with culprit array.
                If not FIXP[n], let n be FIX[n].
                If n<1 or n>ARRAYSIZE[array],
                  cause error 17 with culprit CONS["Out of bounds SETA";n].

                If array is an Integer Array,
                  Let val be FIX[val].
                  Replace the contents of the nth field of array with
                  the results of unboxing val (as an Integer) and return val;
```

elseif array is a Pointer Array,
  Replace the contents of the nth field of
  array with val and return val.

14. HASH ARRAYS

Hash Arrays are objects that provide an efficient way of associating arbitrary objects. To define a Hash Array we must first introduce the notion of a "hash-link".

Definition: A "hash-link" is a meta-object having two fields. The contents of the first field is called the "hash-item" of the hash-link. The contents of the second field is called the "hash-value" of the hash-link. The hash-link represents the association of the hash-item with the hash-value.

Definition: A "Hash Array" of "size" n is an object having n fields, each of which may contain a hash-link. Hash Arrays constitute a distinct class of objects with class name HARRAYP.

Roughly speaking, it is possible to fetch and replace the hash-value associated with some hash-item in a given Hash Array. In addition, subject to certain constraints, it is possible to add a new hash link to a Hash Array, and to remove an old one.

The process of finding the hash-link (if any) in a given Hash Array for some hash-item is called "hash linking" from the hash-item. It is assumed that hash coding is used to make this efficient. The hash coding algorithm used is not specified. It is assumed that the hash coding algorithm implemented will come reasonably close to using all n possible fields before declaring the Hash Array "full".

Initially, the value field of the Literal Atom SYSHASHARRAY shall contain a List Cell whose CAR is a Hash Array and whose CDR is the Floating Point Number 1.5. This Hash Array is used as the user's default Hash Array (i.e., supplied when the Hash Array parameter of the functions below are NIL), and the number indicates the factor by which it is expanded when full[5]. The initial size of the Hash Array is not specified[6]. This Hash Array may not be used to implement any VM facility; it is available only for the user.

The following definition is one of several in this document that involve meta-variables which are understood to take as values the names of other meta-variables and change the denotation of the those (other) meta-variables.

Definition: To "get Hash Array harray", where the meta-variable harray denotes a meta-variable which currently denotes an arbitrary object, means:

"Let obj be the object denoted by (the meta-variable) harray
(note underlining).
If obj is NIL, let harray (note underlining) be SYSHASHARRAY.

---

5 This expansion is not done in the VM but in ERRORX when error 26 is caused.

6 In INTERLISP-10 it is 512.

```
  If LISTP[obj] and HARRAYP[CAR[obj]]:
    let harray (note underlining) be CAR[obj];
  elseif not HARRAYP[obj]:
    cause error 17 with culprit CONS["Arg not hash array";obj]."
```

The reader should understand that if the meta-variable x denotes some non-Hash Array object when the phrase 'Get Hash Array x" is used in a specification, then either an error is caused or else the denotation of x is changed (in particular to some Hash Array).

```
HARRAYP[x]      If x is a Hash Array, return x;
                else, return NIL.

HARRAY[size]    If not FIXP[size], let size be FIX[size].
                Create and return a new Hash Array
                of size size containing no hash-links.

HARRAYSIZE[harray]
                Get Hash Array harray.
                Represent and return as an Integer the size of harray.

PUTHASH[item;val;harray]
                Let origharray be harray.
                Get Hash Array harray.

                If val is NIL,
                  If item is the hash-item of any hash-link in harray,
                  remove that hash-link from harray.
                  Return NIL;
                elseif item is the hash-item of any hash-link in harray,
                  Set the hash-value of that hash-link to val.
                  Return val;
                elseif harray is full,
                  cause error 26 with culprit origharray;
                else:
                  Add a new hash-link to harray, with item as the
                  hash-item and val as the hash-value.
                  Return val.

GETHASH[item;harray]
                Get Hash Array harray.
                If item is the hash-item of any hash-link in harray,
                  return the hash-value of that hash-link;
                else, return NIL.

CLRHASH[harray] Let origharray be harray.
                Get Hash Array harray.
                Remove all hash-links from harray.
                Return origharray.

MAPHASH[fn;harray]
                Get Hash Array harray.
                For every hash-link, x, in harray, compute fn[val;item],
                where item is the hash-item of x and val is the hash-value.
                Return NIL.

REHASH[oldharray;newharray]
                Get Hash Array oldharray.
                CLRHASH[newharray].
                For every hash-link, x, in oldharray, perform
```

```
APPLY*[PUTHASH;item;val;newharray], where item is the
hash-item of x and val is the hash-value.
(The reason APPLY* is used above is only to insure
that a frame extension (cf. Sections 17 and 18) is built for
the call to PUTHASH.  Thus, if newarray is filled and
ERRORX is called by PUTHASH, ERRORX can find the call to
PUTHASH, generate a suitably expanded Hash Array,
initialize it, add the new hash link from item to val which
previously could not be added, and then use the stack
function RETFROM to exit from PUTHASH and continue the
REHASH.)
Return newharray.
```

15. USER DEFINED DATATYPES

The VM allows the user to define new classes of objects. Associated with each such class is a new data type.

In general an object in such a class contains a fixed number of fields (determined according to the definition of the class) each of which may be restricted to contain only certain other kinds of objects or meta-objects. Facilities are provided for declaring the number and type of the fields for a given class, creating objects of a given class, accessing and replacing the contents of the fields of such an object, and interrogating such objects.

In order to define a new class of objects, the user must supply a new data type name and specifications for each of the fields in the objects of the new class.

Definition: A "field specification" is either one of the Literal Atoms POINTER, FIXP, or FLOATP, or else is a proper list of the form (SIGNEDBIT j) or (BIT j), where j is an integer less than the word length of the host machine.

Definition: A field "satisfies a (field specification) spec", if the following relationship holds between spec and the possible contents of the field:

spec	Contents of field
POINTER	Any object
FIXP	Any representable integer (note lower case)
FLOATP	Any representable real (note lower case)
(SIGNEDBIT j)	Any representable integer (note lower case) *whose absolute value is less than 2↑j*
(BIT j)	Any representable non-negative integer (note lower case) whose value is less than 2↑j.

Definition: We say that an object "fields satisfying $spec_1$, $spec_2$, ... $spec_n$" if each of the n fields of the object satisfies a distinct $spec_i$. In this case, we say that the field satisfying $spec_j$ is the jth field (however, nothing is implied about the actual order of the fields in the representation of the object).

Field specifications are used to communicate to the function DECLAREDATATYPE (below) the number of and restrictions on the fields in a new class. DECLAREDATATYPE is free to

arrange the storage allocation for the fields in any way desired by the implementor. DECLAREDATATYPE then returns a proper list of objects, called "field descriptors", which encode the information regarding the position and type of each field. The user can pass such a field descriptor to the functions FETCHFIELD and REPLACEFIELD (below) to access and replace the contents of a given field in an object of the new class. A field descriptor can be any object the implementor wishes to use to carry this information from DECLAREDATATYPE to FETCHFIELD and REPLACEFJELD.

Definition: A "field descriptor" for the jth field in some object is any object the implementor wishes to use which communicates (from the function DECLAREDATATYPE to the functions FETCHFIELD and REPLACEFIELD) sufficient information to allow the contents of the jth field to be accessed and replaced. (Typically, a field descriptor for the jth field must specify the field's type, size, and relative location within the actual representation.)

Convention: If descr is a field descriptor for some field of some object then we may refer to the field as the "descr field" of the object.

```
DECLAREDATATYPE[type;speclst]
                If not LITATOM[type],
                  cause error 17 with culprit CONS["Illegal data type";type];
                elseif type is the data type of an existing class,
                  return GETDESCRIPTORS[type];
                elseif speclst=NIL, cause error 17 with culprit
                CONS["Illegal field specification list";NIL].

                Assume speclst is a non-empty proper list of length n and let
                spec_i, 1=<i=<n, be the elements of speclst.
                If any spec_i, 1=<i=<n, is not a recognized
                field specification, cause error 17 with culprit
                CONS["Illegal field specfication";spec_i].

                Create a new class of objects with data type type
                such that any object of this class shall have n fields
                satisfying spec_1, spec_2, ... spec_n.

                Create and return a new proper list, containing n
                objects, such that the ith element of the proper list
                is a field descriptor for the ith field in objects of data
                type type.

FETCHFIELD[descr;obj]
                If descr is a field descriptor for some data
                type, type, and the type of obj is type:
                  Let spec be the field specification of the descr
                  field of obj.
                  Let val be the contents of the descr field of obj.
                  If spec is POINTER, return val;
                  elseif spec is FIXP or of the form (SIGNEDBIT j)
                  or (BIT j),
                    represent and return as an Integer the integer val;
                  elseif spec is FLOATP,
                    represent and return as a Floating Pointer Number the
                    real val.

REPLACEFIELD[descr;obj;val]
                If descr is a field descriptor for some data type,
                type, and the type of obj is type:
                  Let spec be the field specification for
                  the descr field of obj,
```

```
        If spec is POINTER:
          Let x be val;
        elseif spec is FIXP:
          Let val be FIX[val].
          Let x be the integer represented by val.
        elseif spec is FLOATP:
          Let val be FLOAT[val].
          Let x be the real represented by val.
        elseif spec is of the form (SIGNEDBIT j):
          Let val be FIX[val].
          If the absolute value of val is less than 2↑j.
            let x be the integer represented by val;
          else, truncate val so that its absolute value
          is less than 2↑j (see below), and let x be the
          integer represented by the result;
        elseif spec is of the form (BIT j):
          Let val be FIX[val].
          If val is a non-negative integer less than 2↑j.
            let x be val;
          else,  truncate val so that it is less than 2↑j
          (see below), and let x be the integer represented
           by the result.

        Replace the contents of the descr field of obj with x.
        Return val.
```

Note: We do not define the process of "truncating" an Integer, other than require that it be an operation on Integers that produces an Integer of smaller absolute value. Truncation could be defined to merely produce the low-order digits of the number.

```
NCREATE[type:oldobj]
        If type is not the name of a previously declared
        user data type, cause error 17 with culprit
        CONS["Illegal data type";type].

        Let newobj be a new object of data type
        type, the fields of which have unspecified
        initial contents.

        If TYPENAME[oldobj] is type, replace the contents of
        successive fields in newobj with the objects or
        meta-objects in the corresponding fields of oldobj.

        Return newobj.


GETFIELDSPECS[descr]
        If descr is not the field descriptor of a previously
        declared user data type, type, cause error 17 with culprit
        CONS["Illegal field descriptor";descr].
        Return a field specification which is EQUAL to the one
        supplied for the descr field of objects of type type.

GETDESCRIPTORS[type]
        If not LITATOM[type], let type be TYPENAME[type].
        If type is the name of a user data type,
          create and return a proper list of the field descriptors
          for the n fields of objects of type type, ordered from 1 to n;
        else, return NIL.
```

16. FUNCTIONS AND FUNCTION OBJECTS

In the introductory Sections of this document we introduced several conventions that were central to understanding this document. These conventions concerned the form of function specifications, the notion of meta-variables, the rules governing the use and denotation of meta-variables, and the meaning of the notation "$f[x_1;...x_k]$". These definitions are all at the meta-level in the sense that they tell the reader what the specifications mean.

But the issues involved are central to any programming language. Each of the four meta-concepts above have realization in the VM itself: the representation of INTERLISP programs (function objects), the representation of INTERLISP variables as objects, the processes for accessing and binding INTERLISP variables, and the process for running a function object on given arguments and obtaining the result.

We now begin the discussion of these issues. This Section describes the representation of INTERLISP programs; Section 17 specifies the structures used for binding INTERLISP variables to their values and for keeping track of subroutine (in fact, coroutine) calls; Section 18 specifies the processes for binding and accessing INTERLISP variables and evaluating INTERLISP function objects; Sections 19 and 20 present additional restrictions, refinements and extensions relating to the implementation of the above facilities.

Intuitively, a function object is a program which can be "run" on some arguments to compute some result. A function is just a Literal Atom which has a function object in its function definition field. When a function is applied to a proper list of arguments it is actually the associated function object which is evaluated. Therefore, this Section deals primarily with function objects.

The function objects do not form a disjoint class of objects. For example, some proper lists are function objects. We will discuss the representations of function objects later in this Section.

Every function object must specify how the result of an application is to be computed in terms of the arguments supplied. This specification is done by the "body" of the function object. Of course, with each application the arguments supplied will generally be different. Thus, the computation is specified in terms of Literal Atom "parameter names" which are treated (during the interpretation of the function object by EVAL) as variables representing the actual arguments to be used.

Recall that when we formally specify a VM function we give a list of meta-variables used as parameter names for the VM function specification. The Literal Atom parameter names in a function object are just the realization of these meta-variables. However, nothing is implied about the particular parameter names the implementor should choose when coding the VM functions.

When the function object is applied to some proper list of argument forms, the actual value to be used in place of each parameter name is determined. This raises the following question: Are the parameter names to stand for the values of the argument forms or the forms themselves? In INTERLISP this is determined by a property of the function object being evaluated. The VM allows for two types of function objects: Those of "eval" type are to have their arguments evaluated before the function is activated. Those of "noeval" type are to be activated on the argument forms themselves.

Recall that some of the VM function specifications include the phrase "(NOEVAL)" after the parameter list (cf. AND in Section 5). Formally this means that the function object corresponding to the specification is to have noeval type. All other VM functions are to be of eval type.

In addition to the eval/noeval distinction, INTERLISP provides another property of function objects: A given function object may either take a fixed or variable number of arguments. A function object which takes a fixed number of arguments is called a "spread" type function object, because at application time the arguments are spread across (associated one-to-one with) the parameter names of the function object (with extra arguments ignored and extra parameter names being associated with NIL). If a function object takes a variable number of arguments, then at application time the entire k-tuple of arguments supplied is associated with one parameter name. Such function objects are said to be of "nospread" type.

Those VM function specifications which involve the use of an elipsis ("...") in the parameter list of the specification (cf. AND in Section 5) are to be implemented as nospread function objects. All other VM functions are to be spread function objects.

The parameter names of a function object must be distinct Literal Atoms other than NIL and T, and they must be available to the implementor at application time so that they may be used to set up an association between the names and the values to be used during a particular computation. For spread type function objects, the implementor must be able to obtain an n-tuple of parameter names. This is called the function object's "parameter n-tuple". For nospread type function objects, the implementor must be able to obtain a single parameter name, simply called the function object's "parameter".

The next two Sections present the details of the parameter/value association mechanisms and processes.

The body of a function object specifies a computation in terms of the values of the parameters. The body may be written either in some language which is directly executable by the processor which is running the Virtual Machine, or it may be written as an INTERLISP form which must be interpreted by the Virtual Machine. A function object is called "directly executable" if its body is of the first type.

Most of the functions specified in this document will be implemented as directly executable function objects which have been hand-coded by the implementor. When executed, this code should perform the computations specified in the corresponding VM function specification.

The only kinds of function objects the user himself can create are those that are interpreted. However, most INTERLISP systems provide a compiler which will convert an interpreted function object into a directly executable one. The VM does not require the existence of a compiler. However, it does recognize that a compiler may exist[7].

Finally, some function objects specify a variable binding environment in which the body is to be evaluated. Such function objects are called FUNARGs. FUNARGs have all the properties of other function objects in addition to specifying a Stack Pointer (cf. Section 17) which specifies additional variable bindings to be used during evaluation of the body of the FUNARG.

---

7 The VM puts certain constraints on the compiler if one exists. See Section 20.

Definition: A "function object" is an object with the following properties:

(1) There is a flag specifying whether fnobj is of eval or noeval type.

(2) Either a parameter n-tuple or a parameter is available to the implementor, depending on whether the function is a spread or nospread function object.

(3) There is a body, obtainable by the implementor, which defines some computation either with directly executable code or with a form to be interpreted.

(4) For FUNARG function objects, there is a Stack Pointer which specifies additional variable bindings.

The function objects do not constitute a distinct class of objects. They may be represented in a variety of ways (as discussed below) and may exploit the presence of other data types[8].

Definition: A "SUBR" is a directly executable function object written by the implementor. An "EXPR" is either an interpreted function object or a FUNARG. A "CEXPR" is a directly executable function object generated by the compiler.

We now consider how function objects are represented.

SUBRs may be represented as any objects the implementor desires, provided:

(1) The implementor can recognize such objects as hand-coded function objects.

(2) The object can be determined to be of eval or noeval type (as appropriate).

(3) The parameter n-tuple or parameter (as appropriate) can be obtained from the object (by the implementor).

(4) The body of the function object can be obtained and directly executed.

Additional constraints on the implementation of SUBRs are listed in Section 19.

EXPR function objects other than FUNARGs are represented by proper lists whose first elements are either the Literal Atoms LAMBDA or NLAMBDA. Any List Cell whose CAR is LAMBDA is considered to be a non-FUNARG EXPR function object of eval type. Any List Cell whose CAR is NLAMBDA is considered to be a non-FUNARG EXPR function object of noeval type. If fnobj is a non-FUNARG EXPR function object as above, then fnobj may be assumed to be a proper list of length at least 2. The second element of fnobj determines whether fnobj is a spread or nospread function object. If the second element of fnobj is NIL or a List Cell, fnobj is a spread function object and its second element may be assumed to be a proper list of Literal Atom parameter names: the parameter n-tuple of fnobj is just the n-tuple consisting of the successive elements of the second element of fnobj. If the second element of fnobj is not NIL or a List Cell, then fnobj is a nospread function object and its second element may be

---

8 For example, in INTERLISP-10 Arrays are used to hold directly executable code.

assumed to be a Literal Atom to be used as the parameter of fnobj[9]. The body of an EXPR function object is just the proper list of elements after the second. This proper list is treated as a proper list of forms to be evaluated as specified in the next Section.

CEXPR function objects may be represented as any objects the implementor desires, provided that the restrictions noted in Section 20 are met.

Finally, FUNARG function objects are represented by proper lists. Any List Cell whose CAR is the Literal Atom FUNARG is considered to be a FUNARG function object and may be assumed to be a proper list of length 3. The eval/noeval type, spread/nospread type, parameter n-tuple/parameter and the body of the FUNARG expression are (recursively) those of the second element of the proper list. The third element of the proper list is assumed to be a Stack Pointer. The details of the use of this Stack Pointer are presented in the next Section.

We can summarize the above discussion in three definitions.

Definition: "x is a function object" if either (1) x is a SUBR or CEXPR (which we assume the implementor can determine) or (2) if LISTP[x] and either CAR[x] = LAMBDA or CAR[x] = NLAMBDA (in which case x is a non-FUNARG EXPR) or (3) if LISTP[x] and CAR[x] = FUNARG (in which case x is a FUNARG EXPR).

Definition: "x is of eval type" if x is a function object and one of the following three statements is true: (1) x is a SUBR or CEXPR of eval type (which we assume the implementor can determine) or (2) if x is a non-FUNARG EXPR and CAR[x] = LAMBDA or (3) if x is a FUNARG EXPR and CAR[CDR[x]] is of eval type. "x is of noeval type" if x is a function object and not of eval type.

Definition: "x is a spread function object" if x is a function object and one of the following three statements is true: (1) x is a SUBR or CEXPR spread function object (which we assume the implementor can determine) or (2) if x is a non-FUNARG EXPR and CAR[CDR[x]] is either NIL or a List Cell or (3) if x is a FUNARG EXPR and CAR[CDR[x]] is a spread function object. "x is a nospread function object" if x is a function object and not a spread function object.

```
ARGTYPE[fnobj]  If LITATOM[fnobj], let fnobj be GETD[fnobj].

                If fnobj is not a function object:
                  Return NIL;
                elseif fnobj is of eval/spread type:
                  Return 0;
                elseif fnobj is of noeval/spread type:
                  Return 1;
                elseif fnobj is of eval/nospread type:
                  Return 2;
                elseif fnobj is of noeval/nospread type:
                  Return 3.

FNTYP[fnobj]    If LITATOM[fnobj], let fnobj be GETD[fnobj].

                If fnobj is not a function object:
```

---

9 However, if the Literal Atom is T, the implementor may choose to cause an error when the function object is applied.

```
    return NIL;
  elseif fnobj is a SUBR:
    If fnobj is of eval/spread type, return SUBR;
    elseif fnobj is of noeval/spread type, return FSUBR;
    elseif fnobj is of eval/nospread type, return SUBR*;
    else (fnobj is of noeval/nospread type), return FSUBR*;
  elseif fnobj is a non-FUNARG EXPR:
    If fnobj is of eval/spread type, return EXPR;
    elseif fnobj is of noeval/spread type, return FEXPR;
    elseif fnobj is of eval/nospread type, return EXPR*;
    else (fnobj is of noeval/nospread type), return FEXPR*;
  elseif fnobj is a FUNARG EXPR:
    Return FUNARG;
  else (fnobj is a CEXPR):
    If fnobj is of eval/spread type, return CEXPR;
    elseif fnobj is of noeval/spread type, return CFEXPR;
    elseif fnobj is of eval/nospread type, return CEXPR*;
    else (fnobj is of noeval/nospread type), return CFEXPR*;

SUBRP[fnobj]   If LITATOM[fnobj], let fnobj be GETD[fnobj].

               If fnobj is a SUBR, return T;
               else, return NIL.

EXPRP[fnobj]   If LITATOM[fnobj], let fnobj be GETD[fnobj].

               If fnobj is a List Cell that does not represent
               a SUBR or a CEXPR,
                 return T;
               else, return NIL.
```

Note: EXPRP actually recognizes more than merely the EXPRs, since it will return T on lists that do not have LAMBDA or NLAMBDA in their CARs (as long as such a list does not represent a SUBR or CEXPR.) Since FNTYP actually recognizes only those EXPR function objects described in the text above, it is possible for FNTYP[fn] to be NIL while EXPRP[fn] is T.

```
CCODEP[fnobj]  If LITATOM[fnobj], let fnobj be GETD[fnobj].

               If fnobj is a CEXPR, return T;
               else, return NIL.

ARGLIST[fnobj] If LITATOM[fnobj],
                 Let fnobj be OR[GETD[fnobj];GETP[fnobj;EXPR]].

               If fnobj is not a function object:
                 Cause error 17 with culprit
                 CONS["Args not available:";fnobj];
               elseif fnobj is a FUNARG function object:
                 Return ARGLIST[CAR[CDR[fnobj]]];
               elseif fnobj is a nospread function object:
                 Return the parameter of fnobj;
               elseif fnobj is a spread function object:
                 If fnobj is a (non-FUNARG) EXPR:
                   Return CAR[CDR[fnobj]];
                 else (fnobj is a SUBR or CEXPR):
                   Create and return a new proper list of the
                   successive parameter names in the parameter
                   n-tuple of fnobj.
```

```
NARGS[fnobj]    If LITATOM[fnobj], let fnobj be GETD[fnobj].

                If fnobj is not a function object,
                  return NIL;
                elseif fnobj is a SUBR or CEXPR:
                  If fnobj is of nospread type, return 1;
                  else, return the Integer representing the number
                  of parameters in the parameter n-tuple of fnobj;
                elseif fnobj is a FUNARG function object:
                  Return NARGS[CAR[CDR[fnobj]]];
                else (fnobj is a non-FUNARG EXPR):
                  If fnobj is of nospread type, return 1;
                  else, return the Integer reprsenting the length of
                  the (assumed) proper list CAR[CDR[fnobj]].
```

17. STACK POINTERS

The INTERLISP VM provides a control and access environment structure modeled on the one described by Bobrow and Wegbreit in [1]. The structure is described here, as in [1], as a collection of linked "frames". Although frames are meta-objects, the VM provides a new class of objects, Stack Pointers, as a means of referencing them.

It should be emphasized that the INTERLISP structure is not an exact implementation of the Bobrow-Wegbreit model, but a minor variation of it. The most important difference is that the available frame descriptors are somewhat more restricted and behave differently than in [1].

We present the following (very brief) overview of the INTERLISP stack structure to set the stage for the specifications below and to introduce the terminology to be used.

Function objects and PROG forms share an important property: They are the only objects whose evaluation requires the allocation of storage to hold the values of named local variables. The data structure represented by this allocated storage is called an "access environment" because it is through this structure that the values of variables are accessed. We call function objects and PROG forms "uniform access modules" since their evaluation is responsible for constructing the access environment.

We use the word "activation" to refer to a specific instance of the process of evaluating such a module. An activation of some module requires information in addition to the bindings of named locals. Therefore, associated with each activation of a uniform access module is a meta-object called a "frame extension" which in some sense "contains" all of the access and control information necessary for that activation.

This information includes a pointer to a meta-object which binds variable names to values. Such a meta-object is called a "basic frame" and the field in the frame extension which contains it is called the "blink" or "basic frame link" of the frame extension. Another field in the frame extension, the "alink" or "access link", contains another frame extension which recursively specifies the bindings of all non-local variables. A third field in the frame extension, the "clink" or "control link", points to the frame extension associated with the

activation to which control is to return when the current activation is terminated. Also included in a frame extension is a field which specifies the process associated with the frame (i.e., the computation which is being run in the frame) and which contains storage allocated for unnamed intermediate results and internal control[10].

It is convenient to separate the local binding information (contained in a basic frame) from the more global access and control information (contained in a frame extension). One reason is that basic frames are of fixed size depending upon the number of locals to the module, while the storage allocated to frame extensions depends upon how many temporaries are needed during the computation. Another reason is that it is useful to allow two processes to communicate by sharing variables in a common basic frame.

Every activation of a uniform access module is associated with a basic frame and frame extension. Because a frame extension specifies the basic frame with which it is used, it is sufficient to speak merely of the frame extension associated with any activation.

Convention: When speaking informally we will sometimes use the word "frame" to mean "frame extension".

We can now formally specify the properties of the meta-objects we have introduced above.

Definition: A "binding" is a meta-object containing two fields. The first field is called the "argname" field and contains an object, usually a Literal Atom used as a named local variable. The second is called the "argval" field and contains an object, usually interpreted as the value of the corresponding variable.

Definition: A "basic frame of size n (n>=0)" is a meta-object with the following properties:

(1) There are n bindings, each identified by an integer between 1 and n (provided n>0).

(2) There is a field, called the "frame name" field, which may contain an arbitrary object.

Definition: A "copy of a basic frame" means "a new basic frame of the same size as that of bframe, and containing the same sequence of argnames and argvals and the same frame name."

Definition: "(Literal Atom) var is bound to val in (basic frame) bframe" if there is a binding in bframe with argname var and the last such binding (i.e., the one identified with the largest integer) has argval val. val is said to be the "value" of var in bframe. (It is possible for var to be the argname of two or more bindings in bframe. The last such binding is the only one considered. This is because the search for bindings is usually implemented to start at the back of the basic frame and move up the stack toward the front of the basic frame.)

---

10 In this field we include the "continuation point" for the module when its activation has been suspended for any reason. The continuation point merely indicates where in the module execution is to resume when the activation is continued. Usually the continuation point is just the instruction counter for the code running the process associated with the activation. The Bobrow-Wegbreit paper makes explicit mention of the continuation point field. We include it in the "temporaries" simply because its content is entirely determined by the implementor.

Definition: A "frame extension" is a meta-object with the following properties:

(1) There is a field, called the "blink" field, containing a basic frame.

(2) There is a field, called the "alink" field, containing a frame extension or NIL.

(3) There is a field, called the "clink" field, containing a frame extension or NIL.

(4) There is a field, called the "temporaries" field, containing an unspecified meta-object which specifies all other information specific to the activation associated with the frame extension.

A NIL alink indicates that the top-level values of all non-locals are to be used. A NIL clink indicates that there is no higher process and control cannot return from the frame.

Definition: There is a distinguished frame extension, called the "top-level frame extension" which is associated with the top-level process. The process is specified as follows: "Repeatly execute (without termination) EVALQT[]." The alink and clink of the top-level frame extension are NIL. All other fields are unspecified.

There can be frame extensions other than the top-level one with NIL alink and/or clink. In particular, one cannot necessarily reach the top-level frame extension by simply ascending through the alinks or clinks of successive frames from some starting frame. However, we assume the implementor can always obtain the (original) top-level frame extension.

Definition: The frame (or process or module) from which (or for which) the cpu is currently executing instructions is called the "active" frame (or process or module).

The VM requires the existence of a field, called the "active frame extension" field, which always contains the active frame. This field is available only to the implementor. Except during interrupt processing, the physical machine upon which the Virtual Machine is realized is always executing the instructions for the process associated with the frame in this field. Initially, the active frame extension field contains the top-level frame extension. The function calling and return mechanisms (specified in the next Section) are responsible for maintaining the contents of the active frame extension field.

Definition: The symbol "*actframe*" is an abbreviation for the phrase "the contents of the active frame extension field". Thus, "Let x be *actframe*" is an abbreviation for "Let x be the contents of the active frame extension field." Similary, "Set *actframe* to x" is an abbreviation for "Set the contents of the active frame extension field to x." (The slight pun operating here is quite useful.)

If the active process must invoke a "lower" module then control passes to the lower module (i.e., a new frame extension is built to hold the activation information associated with the lower module and that frame is stored in the active frame extension field) and the previously active process (or frame) is said to be "suspended" awaiting the result of the invoked computation. When a suspended process (or frame) is "reactivated" (with some specified result) then the computation in that module continues where it left off, using the result as the value of the lower module. The point from which processing is to continue is called the "continuation point".

We assume that all control information for the process is maintained in the frame extension. Thus, we no not usually make explicit statements in our specifications regarding saving continuation points before changing the active frame.

Occassionally we will refer to a "copy" of the meta-object in the temporaries field of a frame extension.

Definition: A "copy, tempcop, of the contents, temp, of the temporaries field of a frame extension, frame" is a meta-object containing the same information as temp but which will not be directly affected by continuing the computation in frame. That is, if after obtaining tempcop we allow the computation in frame to continue (which will cause the information in temp to be changed as the computation proceeds) we could get the same behavior (subject to certain obvious but hard to state conditions on the extent to which the computation side-effects the rest of the VM) by replacing the (now modified) temporaries field of frame by tempcop and resuming the computation in frame again.

When a specification constructs a new frame extension without specifying the contents of the temporaries field, the same specification will (almost immediately) make the newly constructed frame the active frame. We mean to imply that the temporaries field of the frame should be so initialized so that the process associated with the frame is that specified after the frame becomes the active frame.

The following concept is analogous to CDR chains from List Cells. It will allow us to talk about the sequence of frame extensions in a chain starting with a given frame extension.

Definition: The "alink chain from frame", where frame is a frame extension or NIL, is that ordered sequence of frame extensions defined recursively as follows: The "alink chain from NIL" is the empty sequence. The "alink chain from frame extension frame" is that sequence obtained by adding frame to the front of the chain of alinks from the alink of frame. We assert the analogous definition of the "clink chain from frame". The "length" of such a chain is just the number of frame extensions in it. Note that if the chain of alinks (or clinks) from frame has non-zero length, then the first element of the chain is always frame.

The manipulation functions specified below insure that no infinite alink/clink chains can be created (i.e., no circular pointer structures through the alink/clink fields can be constructed).

Definition: "var is bound on the access chain from frame" means "some frame extension on the access chain from frame has a basic frame binding var." The "value of var on the access chain from frame" means "the first value of var found by inspecting the successive basic frames on the access chain from frame, starting at frame."

Definition: "(frame extension) x is immediately below (frame extension) y" if y is the alink or clink of x. The relation "below" (applied to frame extensions) is just the transitive closure of "immediately below". We extend this notion to the processes or activations associated with frame extensions as well.

We allow the user to reference a frame extension with a new class of objects:

Definition: A "Stack Pointer is an object having one field which contains a frame extension or a special mark, called the "released mark" (see below).

Frames are usually implemented by allocating storage on a stack of finite length. The stack space occupied by the representation of a frame extension cannot be reclaimed as long as it is possible for control to reach it. In particular, it cannot be reclaimed if the user has a Stack Pointer which references the frame extension. Therefore, we allow the user to explicitly sever the link between a Stack Pointer and the frame extension it contains. This is done by depositing a special meta-object, called the "released mark", in the Stack Pointer. The

function RELSTK does this. Of course, whether the storage associated with the frame extension can then be reclaimed still depends upon whether it is possible for control to reach the frame extension.

Definition: The "released mark" is a special meta-object which is distinct from any frame extension and which can be deposited in the field of a Stack Pointer. Its presence in such a field indicates that the Stack Pointer no longer references a frame extension.

Section 18 will formally describe the functions which are responsible for activating modules and interpreting the contents of frames as environments. We now proceed with the specification of the functions which manipulate Stack Pointers as data-objects.

Because it is inconvenient (and causes the allocation of additional storage) to obtain a Stack Pointer to reference a particular frame extension, we allow a variety of objects to describe certain frame extensions. In general, T and NIL describe the top-level and current frame extensions (respectively). Other Literal Atoms describe the lowest frame extension with that Literal Atom as its frame name, and Integers describe the frame extension a given distance down the alink or clink chain (depending on the algebraic sign). We formalize this below in another definition which takes as a parameter a meta-variable name and assigns that meta-variable a new value.

Definition: The phrase "get frame extension x", where x denotes a meta-variable which denotes an object means:

```
"Let obj be the object denoted by x.

Let actstkptr be a Stack Pointer containing *actframe*.
(Below we will call VM functions to interrogate the
stack and, since we cannot call such a function on *actframe*
directly (since frame extensions are meta-objects) we must assume the
existence of the redundant stack pointer actstkptr.)

If STACKP[obj]:
  If RELSTKP[obj], cause error 30 with culprit obj;
  else, let obj be the frame extension contained in obj;
elseif obj = T, let x be the top-level frame extension;
elseif obj = NIL, let x be *actframe*;
elseif LITATOM[obj]:
  If STKPOS[obj;-1;actstkptr], let x be the frame extension
  contained in STKPOS[obj;-1;actstkptr];
  else, cause error 19 with culprit obj;
elseif obj is a Number:
  If STKNTH[obj;actstkptr], let x be the frame extension
  contained in STKNTH[obj;actstkptr];
  else, cause error 19 with culprit obj;
else, cause error 19 with culprit obj."
```

Note that if we say "Get *frame extension frame*" where *frame* is some object such as a Stack Pointer, NIL, T, a function name or an Integer, then thereafter frame denotes a (meta-object) frame extension (or else an error was caused). The frame extensions described by objects other than T and Stack Pointers are defined relative to the frame extension which is running the stack function which uses the "get frame extension" notation (i.e., the frame extension which is *actframe*). This is at variance with the Bobrow-Wegbreit model which computes these frame extensions relative to the frame which called the stack function. This means that the frame descriptors here behave somewhat differently than in the Bobrow-Wegbreit model.

In order to avoid creating Stack Pointers most stack functions can be made to reuse an existing Stack Pointer when one is given. We introduce the following definition to make this convenient.

Definition: To "return a Stack Pointer containing frame (using stkptr)", where frame is a frame extension and stkptr is an arbitrary object, means:

```
"If STACKP[stkptr]:
  Set the contents of stkptr to frame.
  Return stkptr;
else, create and return a new Stack Pointer containing frame."
```

Note that after replacing the contents of stkptr with frame, the storage associated with the previous contents of stkptr may be subject to reclamation.

```
STACKP[x]       If x is a Stack Pointer, return x;
                else, return NIL.

STKPOS[name;n;frame;stkptr]
                Get frame extension frame.
                RELSTK[stkptr].
                If n is NIL, let n be -1;
                elseif not FIXP[n], let n be FIX[n].
                If n=0, let n be 1.
                If n<0, let chain be the clink chain from frame;
                elseif n>0, let chain be the alink chain from frame.
                If there are at least |n| elements of chain
                containing basic frames with frame name name:
                  Let newframe be the |n|th such element.
                  If newframe is *actframe*,
                    cause error 19 with culprit NIL.
                  Return a Stack Pointer containing newframe (using stkptr);
                else, return NIL.

STKNTH[n;frame;stkptr]
                Get frame extension frame.
                RELSTK[stkptr].
                If n is NIL, let n be -1;
                elseif not FIXP[n], let n be FIX[n].
                If n=0, let n be 1.
                If n<0, let chain be the clink chain from frame;
                elseif n>0, let chain be the alink chain from frame.
                If n=0:
                  If frame is *actframe*,
                    cause error 19 with culprit NIL;
                  else, return a Stack Pointer containing frame (using stkptr);
                elseif the length of chain does not exceed |n|,
                  return NIL;
                else, return a Stack Pointer containing the |n|+1st
                element of chain (using stkptr).

MKFRAME[frame;alink;clink;flg;stkptr]
                Get frame extension frame.
                RELSTK[stkptr].
                If alink is NIL, let alink be the alink of frame;
                else, get frame extension alink.
                If clink is NIL, let clink be the clink of frame;
                else, get frame extension clink.

                Let bframe be the basic frame of frame.
```

```
                If flg:
                  Let bframe be a copy of the basic frame bframe.

                Create a new frame extension, newframe, such that:
                  The blink field of newframe contains bframe.
                  The alink field of newframe contains alink.
                  The clink field of newframe contains clink.
                  The temporaries field of newframe contains a copy of the
                  meta-object in the temporaries field of frame.

                Return a Stack Pointer containing newframe (using stkptr).

STKNARGS[frame] Get frame extension frame.
                Represent and return as an Integer the size
                of the basic frame of frame.

STKARGNAME[n;frame]
                Get frame extension frame.
                Let bframe be the basic frame of frame.
                If LITATOM[n]
                  If there is a binding in bframe with argname n,
                    return n;
                  else, cause error 19 with culprit n;
                else:
                  If not FIXP[n], let n be FIX[n].
                  If n>0 and there are at least n bindings in bframe,
                    return the argname of the nth binding in bframe;
                  else, cause error 19 with culprit n.

STKARG[n;frame] Get frame extension frame.
                Let bframe be the basic frame of frame.
                If LITATOM[n]
                  If there is a binding in bframe with argname n,
                    return the argval of the last such binding;
                  else, cause error 19 with culprit n;
                else:
                  If not FIXP[n], let n be FIX[n].
                  If n>0 and there are at least n bindings in bframe,
                    return the argval of the nth binding in bframe;
                  else, cause error 19 with culprit n.

SETSTKARGNAME[n;frame;name]
                Get frame extension frame.
                Let bframe be the basic frame of frame.
                If LITATOM[n]
                  If there is a binding in bframe with argname n:
                    Set the argname field of the last such binding to name.
                    Return name;
                  else, cause error 19 with culprit n;
                else:
                  If not FIXP[n], let n be FIX[n].
                  If n>0 and there are at least n bindings in bframe:
                    Set the argname field of the nth binding in bframe to name.
                    Return name;
                  else, cause error 19 with culprit n.

SETSTKARG[n;frame;val]
                Get frame extension frame.
                Let bframe be the basic frame of frame.
                If LITATOM[n]
                  If there is a binding in bframe with argname n:
```

```
                    Set the argval field of the last such binding to val.
                    Return val;
                  else, cause error 19 with culprit n;
                else:
                  If not FIXP[n], let n be FIX[n].
                  If n>0 and there are at least n bindings in bframe:
                    Set the argval field of the nth binding in bframe to val.
                    Return val;
                  else, cause error 19 with culprit n.

STKNAME[frame]  Get frame extension frame.
                Return the contents of the frame name field
                of the basic frame of frame.

RELSTKP[stkptr] If STACKP[stkptr] and stkptr contains the released mark,
                  return stkptr;
                else, return NIL.

RELSTK[stkptr]  If STACKP[stkptr],
                  set the contents of stkptr to the released mark.
                Return stkptr.

CLEARSTK[flg]   If flg:
                  Create and return a new proper list of
                  all existing Stack Pointers that do not
                  contain the released mark;
                else:
                  For every existing Stack Pointer, x, that does
                  not contain a released mark,
                    set the contents of x to the released mark.
                  Return NIL.
```

Definition: A "copy of the alink chain of startframe to endframe", where startframe and endframe are frame extensions and endframe is in the alink chain of startframe, means:

```
"If startframe=endframe:
  A new frame extension with:
    Blink set to a copy of the basic frame of startframe.
    Alink set to the alink of startframe.
    Clink set to the clink of startframe.
    Temporaries set to a copy of the temporaries of startframe.
else:
  A new frame extension with:
    Blink set to a copy of the basic frame of startframe.
    Alink set to a copy of the alink chain of the alink
    of startframe to endframe.
    Clink set to the clink of startframe.
    Temporaries set to a copy of the temporaries of startframe."

COPYSTK[startframe;endframe]
                Get frame extension startframe.
                Get frame extension endframe.
                If endframe is not in the alink chain from startframe:
                  Cause error 19 with culprit NIL.
                Let newframe be a copy of the alink chain from
                startframe to endframe.
                Create and return a new Stack Pointer containing newframe.

FRAMESCAN[var;frame*]
                Get frame extension frame.
                If the basic frame of frame contains a binding
```

```
                with argname var:
                  Let i be the integer associated with the
                  last such binding in the basic frame of frame.
                  Represent and return as an Integer the integer i;
                else, return NIL.

STKSCAN[var;frame;oldptr]
                Get frame extension frame.
                RELSTK[oldptr].
                If there is a frame extension on the alink chain
                from frame which has a binding with argname var:
                  Let x be the first such frame extension.
                  Return a stack pointer containing x (using oldptr);
                else, return NIL.

STKNTHNAME[n;frame]
                Let actstkptr be a Stack Pointer containing *actframe*.
                Return STKNAME[STKNTH[n;frame;actstkptr]].
```

Note: This function is so frequently used that it is best implemented so as to avoid the unnecessary construction of a new Stack Pointer by STKNTH.

18. EVALUATION

This Section specifies the INTERLISP interpreter and how the process of interpretation interacts with the access and control stack described in the previous Section.

The VM allows the user to discover certain information about the internal state of several VM functions. These functions are EVAL, APPLY, COND, PROG, PROGN and PROG1. This state information is maintained in several fields associated with particular activations of the functions. We call these fields the "blip fields".

Definition: A "blip field" is a field used to store information regarding the internal state of the VM functions EVAL, APPLY, COND, PROG, PROGN and PROG1. These functions are called the "blip-using functions". There are four types of blip fields. Each type of blip field is named by a Literal Atom and may contain any object. The name of each type of blip field and its usual contents is given below:

blip field name	usual contents
FN	any function or function object
ARGVAL	any object
FORM	any form
TAIL	any proper list of forms (or clauses -- see COND)

There may be at most one *FN*, *FORM*, and *TAIL* blip field for any activation of a blip-using function. There are generally a variable number of *ARGVAL* fields.

Not every activation of a blip-using function will necessarily have a blip field associated with it. The specifications of the blip-using functions explicitly deal with the allocation and manipulation of blip fields. The functions BLIPSCAN, BLIPVAL, and SETBLIPVAL (defined in Section 19) allow the user to access and replace the contents of these fields.

Basically, these fields just represent those temporaries necessary to actually implement the blip-using functions. For example, when EVAL is computing the values of the arguments to be supplied to some function, it uses the *FN* field to hold the function which will (ultimately) be evaluated, the *ARGVAL* fields to hold the argument values already computed, the *FORM* field to hold the argument form currently being considered, and the *TAIL* field to hold the proper list of argument forms not yet considered. As we will discuss in the next Section, it is possible to implement the blip-using functions in such a way that these fields are literally local variables bound in a basic frame.

To make it convenient to refer to the contents of these fields we make the following convention.

Definition: The symbol "*fn*", used in the context of some activation of a blip-using function, is an abbreviation for "the contents of the *FN* field associated with this activation". We make the analogous conventions for *form* and *tail*. (We will not need such a convention for the *ARGVAL* fields.)

We now begin the formal account of how a form is evaluated. This account essentially depends on two fundamental issues: the way local variables are bound and the way functions are called and return results. These concepts are formalized below.

We first specify how the variables in a module are bound to their values for a particular activation of the module. This is done by constructing a basic frame from the module's name, the associated function object, and a proper list of forms supplying the arguments. This procedure is also responsible for associating and maintaining the blip fields for activations of EVAL and APPLY.

Definition: To construct a "new basic frame, x, from *fnname*, *fnobj*, and *arglist*", where x denotes a meta-variable, fnname is a Literal Atom, fnobj is a function object, and arglist is a proper list of k forms, the following procedure is followed:

```
"Create and associate a new *FN* field, a new *FORM* field
and a new *TAIL* field with this activation of the blip-using function
using this definition.
Set *fn* to fnname.

If fnobj is a nospread function object of noveval type:
  Let k be 1.
  Create and associate a new *ARGVAL* field with this
  activation.
  Replace the contents of this *ARGVAL* field with arglist.
else (we must consider the successive elements of arglist):
  Set *tail* to arglist.
  For i from 1 to k do the following:
    Set *form* to CAR[*tail*].
    If fnobj is of noeval type, let val be *form*;
    else, let val be EVAL[*form*].
    Create and associate a new *ARGVAL* field with this
    activation, and replace the contents of this field by val.
    Set *tail* to CDR[*tail*].


(We have now stored all of the argument values in k new *ARGVAL* fields and are
prepared to build a suitable basic frame.)
```

```
In the following, consider the k *ARGVAL* fields in the reverse order of their
creation, i.e., let the 1st *ARGVAL* field be the one most recently created and
associated with this activation, and the kth *ARGVAL* field be the one first
created and associated with this activation.

(We must now inspect the contents of the *FN* field in case it has been modified
(with SETBLIPVAL) during an interrupt or a lower call to EVAL.)

Let fnname be *fn*.
If fnname is a function or function object:
  (We only actually build a basic frame, bframe, if fnname is
  still a function or function object.  If the contents of the
  *FN* field was replaced by some other kind of object,
  no basic frame is constructed and special action is taken
  by the procedure which employed this definition.)

  If fnname is a function:
    Let fnobj be GETD[fnname];
  elseif fnname is a function object:
    Let fnobj be fnname.

  If fnobj is a nospread function object:
    Let param be the parameter of fnobj.
    If fnobj is of noeval type:
      Create a new basic frame, bframe, of size 1.
      Let the frame name of bframe be fnname.
      Let the binding in bframe have argname param and argval the
      contents of the 1st *ARGVAL* field associated with this
      activation (or NIL if k is 0)..
    else (fnobj is of eval type):
      Create a new basic frame, bframe, of size k+1.
      Let the frame name of bframe be fnname.
      For i from 1 to k do the following:
        Replace the argname field of the ith binding in
        bframe by some unspecified object or meta-object
        other than a Literal Atom.
        Replace the argval field of the ith binding in
        bframe by the contents of the k-i+1st *ARGVAL* field.
      Let the k+1st binding in bframe have argname param and argval the
      representation of k as an Integer.
    else (fnobj is a spread function object):
      Let n be the number of parameter names in the parameter n-tuple
      of fnobj and let param_i (1=<i=<n) be the ith component of this n-tuple.
      Create a new basic frame, bframe, of size n.
      Let the frame name of bframe be fnname.
      For i from 1 to n do the following:
        Replace the argname field of the ith binding in
        bframe by param_i.
        If i =< k:
          Replace the contents of the argval field of
          the ith binding in bframe by the contents of the
          k-i+1st *ARGVAL* field associated with this activation.
        else:
          Replace the contents of the argval field of the ith
          binding in bframe by NIL.
  Let x be bframe."
```

Note that after a use of the phrase "construct a new basic frame, basic, from fn, fnobj, and arglist", new blip fields will be associated with the frame extension of the function concerned, and, unless the *FN* field does not contain a function or function object, the meta-variable basic will denote a new basic frame constructed as described above.

The existence and use of the blip fields (and the existence of the function BLIPVAL) allow DWIM to discover the context in which certain errors occur. In particular, DWIM can find out the function which is waiting to be called (*FN*), the values of the argument forms already evaluated (the *ARGVAL* fields), the argument form currently being evaluated (*FORM*), and the proper list of remaining argument forms (*TAIL*). The function SETBLIPVAL allows DWIM to alter these fields if the error is diagnosed, so that the interpreter continues with the evaluation as if no error had occurred.

Continuing with the discussion of bound variables, we now make two useful definitions regarding references to variables.

Definition: A Literal Atom is said to be a "local variable" of a function object or PROG form if the Literal Atom is referenced as a variable (see the specifications of EVAL) in the body of the function object or PROG form, and is always bound in a basic frame at or below the one in which the function object or PROG form is evaluated. In particular, the local variables of a function object include its parameter names and the locals of PROG, LAMBDA, and NLAMBDA expressions appearing (structurally) within the body.

Definition: A Literal Atom is said to be a "non-local" or "free" variable of function object or PROG form if it is referenced as a variable in that form but is not a local variable of the function object or PROG.

We next formalize the notions of function call and return.

Definition: The "result of evaluating (or calling) fnname on arglist", where fnname is a function or function object and arglist is either a proper list of forms or a basic frame, is the object denoted by the meta-variable result after the following computation:

```
"If LITATOM[fnname]:
  Let fnobj be GETD[fnname];
else (fnname is a function object):
  Let fnobj be fnname.

If fnobj is a FUNARG function object:
  Let blink be a new basic frame with frame name NIL and 0 bindings.
  Let stkptr be CAR[CDR[CDR[fnobj]]].
  If not STACKP[stkptr] or RELSTKP[stkptr],
    cause error 19 with culprit stkptr.
  Let alink be the frame extension contained in stkptr;
else:
  If arglist is a basic frame:
    Let bframe be arglist;
  else (arglist is a proper list of forms):
    (In this case we must construct the appropriate basic frame.)
    Construct a new basic frame, bframe, from fnname, fnobj, and arglist.
    (We must now treat the contents of the *FN* field as though it
    is the function or function object to be applied.)
    Let fnname be the contents of the *FN* field created and associated with
    this activation during the construction of bframe.
    If fnname is not a function and is not a function object:
      Let argvals be a new proper list of the contents of the *ARGVAL*
      fields associated with this activation, in the order of their
      creation.
      Return FAULTAPPLY[fnname;argvals];
    elseif LITATOM[fnname]:
      Let fnobj be GETD[fnname];
    else (fnname is a function object):
```

```
      Let fnobj be fnname.
    Let blink be bframe.
    Let alink be *actframe*.

  Let frame be a new frame extension such that:
    The blink field of frame contains blink.
    The alink field of frame contains alink.
    The clink field of frame contains *actframe*;

  Set *actframe* to frame.
  If fnobj is a FUNARG function object:
    Let result be the result of evaluating CAR[CDR[fnobj]] on arglist.
  elseif fnobj is directly executable:
    Let result be the result of executing the instructions
    in the body of fnobj;
  else (fnobj is not directly executable):
    For successive elements, form, in the (assumed proper list) body of fnobj:
      Let result be EVAL[form].

  Set *actframe* to the clink of *actframe*."
```

The preceding definition is only used by EVAL and APPLY and consequently, use of those functions (or their variants) are the only way the user can evaluate forms or apply functions. Thus, user calls to VM functions always have frames associated with them.

The action of the interpreter, EVAL, on forms other than Literal Atoms, Numbers, and List Cells is determined by a table, called the "EVAL table":

Definition: The "EVAL table" is a meta-object which has as many fields as there are existing data types. Each field is identified by one the Literal Atom data type names, and may contain any object. Note that the size of the EVAL table increases with each new user defined data type.

When EVAL encounters a form whose data type is other than one of those mentioned above, the data type's entry in the EVAL table determines the actions of EVAL. If the entry is T, EVAL will return the form as its value. If the entry is a function object, EVAL will apply the function object to the form and return the result as the form's value. The function DEFEVAL (specified below) allows the user to modify the entries in this table.

The initial configuration of the EVAL table is that the entries for LITATOM, FIXP, FLOATP, and LISTP are unspecified (they are never inspected since the behavior of EVAL on such forms is built-in) and the entry for every other existing data type is T. Whenever a new data type is created by the user, the associated field in the EVAL table is initialized to T.

```
EVAL[form]      If LITATOM[form]:
                  If form = NIL or form = T, return form;
                  else:
                    (We say that form "has been referenced as a
                    variable" in any form whose evaluation might
                    arrive at this point.)
                    If form is bound on the access chain from *actframe*,
                      Return the binding of form;
                    elseif GETTOPVAL[form] = NOBIND:
                      Return FAULTEVAL[form];
                    else, return GETTOPVAL[form];
                elseif FIXP[form] or FLOATP[form]:
                  Return form;
```

```
        elseif LISIP[form]:
          (Note:  form is assumed to be a proper list.)
          If CAR[form] is a function or function object:
            Return the result of evaluating CAR[form] on CDR[form];
          else, return FAULTEVAL[form].
        else (form is other than a Literal Atom, Number, or List Cell):
          Let type be TYPENAME[form].
          Let fnobj be the contents of the type field in the EVAL table.
          If fnobj = T, return form;
          else, return APPLY[fnobj;CONS[form;NIL]].
```

Note: We assume that EVAL is implemented as a directly executable function (or else "evaluate *fnobj* on *arglist*" would never be a terminating process on EXPR function objects).

```
EVALV[var;frame]
        (We asssume var to be a Literal Atom.)
        Get frame extension frame.
        If there is a frame extension in the alink chain of
        frame with a basic frame containing a binding
        with argname var:
          Return the contents of the argval field of the last
          such binding in the first such basic frame;
        else return GETTOPVAL[var].

SET[var;val]    If there is a frame extension, x, in the access
                chain from *actframe*, which binds var:
                  Let bframe be the basic frame of the first such
                  frame extension.
                  Set the argval field of the last binding of var
                  in bframe to val.
                  Return val;
                else, SETTOPVAL[var;val].

SETQ[var;val] (NOEVAL)
        Return SET[var;EVAL[val]].

EVALA[form;alist]
        (We assume alist is a proper list of List Cells.)
        Let n be the length of alist.
        Construct a new basic frame, bframe, of size n
        such that the frame name is NIL and the ith binding
        has argname u and argval v, where u and v are the
        CAR and CDR (respectively) of the ith element of
        alist (1 =< i =< n).

        Construct a new frame extension, frame, such that:
          The blink field of frame contains bframe.
          The alink field of frame contains *actframe*.
          The clink field of frame contains *actframe*.

        Set *actframe* to frame.
        Let val be EVAL[form].
        Set *actframe* to the clink of *actframe*.
        Return val.

DEFEVAL[type;fnobj]
        If type is not the name of an existing data type:
          Cause error 33 with culprit type;
        elseif type is one of the Literal Atoms LITATOM,
```

```
                FIXP, FLOATP, or LISTP:
                  Cause error 33 with culprit type;
                elseif fnobj is NIL:
                  Return the contents of the type field of the EVAL table;
                else:
                  Let oldval be the contents of the type field of
                  the EVAL table.
                  Set the contents of the type field of the EVAL table
                  to fnobj.
                  Return oldval.

FUNCTION[form;env] (NOEVAL)
                If env=NIL, return form;
                elseif STACKP[env], return LIST[FUNARG;form;env];
                elseif LISTP[env]:
                  (We assume env to be a proper list of n Literal Atoms.)
                  Construct a new basic frame, bframe, with frame name
                  FUNARG and containing n bindings, such that the
                  ith binding, (1=<i=<n) has argname u and argval EVALV[u],
                  where u is the ith element of env.

                  Construct a new frame extension, frame, such that:
                    The blink field of frame contains bframe.
                    The alink field of frame contains *actframe*.
                    The clink field of frame contains NIL.

                  Construct a new Stack Pointer, stkptr, containing frame.

                  Return LIST[FUNARG;form;stkptr];
                else, cause error 27 with culprit env.

ENVEVAL[form;alink;clink;aflg;cflg]
                Let origalink be alink.
                Let origclink be clink.
                Get frame extension alink.
                Get frame extension clink.
                Let frame be a new frame extension such that:
                  The blink field of frame contains a basic frame
                  containing no bindings and frame name NIL.
                  The alink field of frame contains alink.
                  The clink field of frame contains clink.

                If STACKP[origalink] and aflg, RELSTK[origalink].
                If STACKP[origclink] and cflg, RELSTK[origclink].

                Save the continuation point for the active frame in
                *actframe* so that, if *actframe* is ever reactivated
                with result x, this activation will return x.

                Set *actframe* to frame.
                Let val be EVAL[form].
                If the clink of *actframe* is NIL,
                  cause error 3 with culprit val.
                Set *actframe* to the clink of *actframe*.
                Reactivate the process in the clink of *actframe*
                with result val.


RETFROM[frame;val;flg]
```

```
                Let origframe be frame.
                Get frame extension frame.

                If STACKP[origframe] and flg, RELSTK[origframe].

                If the clink of frame is NIL,
                  cause error 3 with culprit val.
                Set *actframe* to the clink of frame.
                Reactivate the process in the clink of frame
                with result val.

RETTO[frame;val;flg]
                Let origframe be frame.
                Get frame extension frame.

                If STACKP[origframe] and flg, RELSTK[origframe].

                Set *actframe* to frame.
                Reactivate the process associated with frame
                with result val.


APPLY[fn;arglist]
                (Assume arglist is a proper list.)
                If fn is not a function or function object,
                  return FAULTAPPLY[fn;arglist].

                Return the result of evaluating fn on arglist (treating
                fn as though it were of noeval type).


APPLY*[fn;arg1;arg2;...argn]
                Return APPLY[fn;LIST[arg1;arg2;...argn]].
```

Note: APPLY* is used so frequently it is best implemented so as to avoid creating a proper list of the arg_i's when possible. For noeval/nospread functions it is not possible to avoid creating the list.

```
ENVAPPLY[fn;arglist;alink;clink;aflg;cflg]
                Let origalink be alink.
                Let origclink be clink.
                Get frame extension alink.
                Get frame extension clink.
                Let frame be a new frame extension such that:
                  The blink field of frame contains a basic frame containing
                  no bindings and frame name NIL.
                  The alink field of frame contains alink.
                  The clink field of frame contains clink.

                If STACKP[origalink] and aflg, RELSTK[origalink].
                If STACKP[origclink] and cflg, RELSTK[origclink].

                Save the continuation point for the active frame in *actframe*
                so that, upon reactivation with result x, this activation
                of ENVAPPLY will return x.

                Set *actframe* to frame.
                Let val be APPLY[fn;arglist].
```

```
                If the clink of *actframe* is NIL,
                  cause error 3 with culprit val.
                Set *actframe* to the clink of *actframe*.
                Reactivate the process associated with the clink
                of *actframe* with result val.

ARG[var;n] (NOEVAL)
                Let n be EVAL[n].
                If var is not bound on the access chain from
                *actframe*, cause error 27 with culprit var.
                Let k be the value of var on the access chain
                from *actframe*, and let bframe be the basic
                frame containing the binding of var to k.
                Let size be the size of bframe.
                If k /= size-1, cause error 27 with culprit var;
                elseif n < 1 or n > k, cause error 27 with culprit n.
                Return the argval of the nth binding in bframe.

SETARG[var;n;val] (NOEVAL)
                Let val be EVAL[val].
                Let n be EVAL[n].
                If var is not bound on the access chain from *actframe*,
                cause error 27 with culprit var.
                Let k be the value of var on the access chain
                from *actframe*, and let bframe be the basic frame
                containing the binding of var to k.
                Let size be the size of bframe.
                If k /= size-1, cause error 27 with culprit var;
                elseif n < 1 or n > k, cause error 27 with culprit n.
                Set the argval of the nth binding in bframe to val.
                Return val.

COND[clause1;clause2;...clausen] (NOEVAL)
                (Each clausei is assumed to be a proper list of forms
                and is called a "clause".)
                Associate new *FORM* and *TAIL* fields with *actframe*.
                Let *tail* be the contents of the first argval
                field of *actframe*.
                (This will be the proper list of clausei's in
                the COND form being evaluated).
                Until *tail* is NIL do the following:
                  Let *form* be CAR[CAR[*tail*]].
                  Let val be EVAL[*form*].
                  If val is not NIL:
                    Let *tail* be CDR[CAR[*tail*]].
                    Until *tail* is NIL do the following:
                      Let *form* be CAR[*tail*].
                      Let val be EVAL[*form*].
                      Let *tail* be CDR[*tail*].
                    Return val.
                  Let *tail* be CDR[*tail*].
                (If control reaches this point, the CAR of
                each clausei EVALd to NIL.)
                Return NIL.

PROG[localvars;form1;form2;...formn] (NOEVAL)
                (Note:  localvars is assumed to be a proper list.)
                Let progbody be the argval of the first binding in
                the basic frame associated with *actframe* (this will
                be the proper list of arguments to the PROG form
                being evaluated).
```

```
Mark the temporaries field of *actframe* so that it can
be recognized as a frame in which an activation of
PROG is running (see Note below).

Let k be the length of localvars.
For i from 1 to k do the following:
  Let x be the ith element of localvars.
  If x is a Literal Atom:
    Let var_i be x.
    Let val_i be NIL;
  else (x is assumed to be a proper list with
  a Literal Atom in its CAR):
    Let var_i be CAR[x].
    Let val_i be EVAL[CAR[CDR[x]]].
Construct a new basic frame, bframe, with frame name *PROG*LAM
and containing k bindings such that the ith binding, 1=<i=<k,
binds var_i to val_i.

Construct a new frame extension, frame, such that:
  The blink field of frame contains bframe.
  The alink field of frame contains *actframe*.
  The clink field of frame contains *actframe*.

Set *actframe* to frame.
Associate new *FORM* and *TAIL* fields with *actframe*.
Let *tail* be CDR[progbody].

Until *tail* is NIL do the following:
  Let *form* be CAR[*tail*].
  If *form* is not a Literal Atom, EVAL[*form*].
  Let *tail* be CDR[*tail*].

Set *actframe* to the clink of *actframe*.
Return NIL.
```

Note: The following two functions, GO and RETURN, are used to modify the flow of control in PROG. They do this by inspecting the stack and reactivating the appropriate frames (possibly modifying the blip fields used by PROG). Thus, it is important that these two functions be able to recognize PROG frames. It is not sufficient to assume that the frame name of such frames will always be PROG. This is because some of the high-level functions in the INTERLISP user support facilities (e.g., the ADVISE feature) may deposit the function object associated with PROG in the function definition field of another Literal Atom and activate it by applying that Literal Atom instead of PROG.

```
GO[label] (NOEVAL)
        If there is a frame extension in the control chain from
        *actframe* that is marked as a PROG frame (cf. PROG above)
        and label is an element of the (assumed) proper list
        in the first argval field of its basic frame:
          Let progframe be the first such frame extension.
          Let progbody be the contents of the first argval
          field in the basic frame of progframe;
        else, cause error 8 with culprit label.

        (If progframe exists then the call to EVAL running in the
        frame named frame in the specification of PROG above
        is suspended while waiting for the results of the computation
        that involved this application of GO.)
```

Let lowerprogframe be the frame immediately under progframe (i.e., the frame called frame in the specification of PROG above).

Let progtail be the terminal sublist of progbody starting with the first occurrence of label in progbody.

Set the contents of the *TAIL* blip field in lowerprogframe to progtail.

Reactivate the process in lowerprogframe with result NIL (i.e., continue the "Until" loop running in lowerprogframe just as though the call to EVAL had returned NIL).

RETURN[val]

If there is a frame extension, frame, in the control chain from *actframe* that is marked as a PROG frame (cf. PROG above):
  Let frame be the first such frame.
  If the clink of frame is NIL,
    cause error 3 with culprit frame.
  Set *actframe* to the clink of frame.
  Reactivate the computation associated with *actframe* with result val.
else, cause error 3 with culprit NIL.

PROGN[$form_1$:$form_2$:...$form_n$] (NOEVAL)

Let val be NIL.
Associate new *FORM* and *TAIL* fields with *actframe*.
Let *tail* be the contents of the first argval field in the basic frame of *actframe* (this will be the proper list of forms supplied as arguments to the PROGN form being evaluated).
Until *tail* is NIL do the following:
  Let *form* be CAR[*tail*].
  Let val be EVAL[*form*].
  Let *tail* be CDR[*tail*].
Return val.

PROG1[$form_1$:$form_2$:...$form_n$] (NOEVAL)

Associate new *FORM* and *TAIL* fields with *actframe*.
Let *tail* be the contents of the first argval field in the basic frame *actframe* (this will be the proper list of forms supplied as arguments to the PROG1 form being evaluated).
If *tail* is NIL, return NIL.
Let *form* be CAR[*tail*].
Let val be EVAL[*form*].
Let *tail* be CDR[*tail*].
Until *tail* is NIL do the following:
  Let *form* be CAR[*tail*].
  EVAL[*form*].
  Let *tail* be CDR[*tail*].
Return val.

BACKTRACE[$frame_1$:$frame_2$:flags]

Get frame extension $frame_1$.
If $frame_2$ is NIL, let $frame_2$ be T.
Get frame extension $frame_2$.

If not FIXP[flags], let flags be FIX[flags].
Let n be some integer greater than 3 and let b_0, b_1, .. b_n be the binary digits so that

```
flags = b0 + b1*2^1 + ... + bn*2^n.

If b4=1, let chain be that subsequence of
the alink chain from frame1 to (and including)
frame2 (or, if frame2 is not in frame1's alink chain,
the top-most frame extension in frame1's alink chain)
else, let chain be that subsequence of the clink
chain from frame1 to (and including) frame2 (or, if
frame2 is not in frame1's clink chain, the top-most
frame extension in frame1's clink chain).

For each frame, frame, in chain, do the following:
  Let fn be the frame name of the basic frame of frame;
  Write (to the terminal and in any format desired)
  the following information (provided the conditions
  on the bi's and fn are satisfied):
    (1) fn (provided b3=0).
    (2) both components of each binding in the
        basic frame of frame (provided either
          (a) b0=1 and not SUBRP[fn]
                or
          (b) b2=1 and SUBRP[fn]).
    (3) names and values of all temporaries used
        by EVAL (provided b1=1 and b2=0).
    (4) names and values of all temporaries
        (whether used by EVAL or not) whose values can be
        meaningly displayed (provided b2=1).

Return T.
```

19. RESTRICTIONS ON THE IMPLEMENTATION OF VM FUNCTIONS

There are several important points which should be made relating to the actual implementation of the VM functions specified in this document.

The first concerns the CONS count, and the Large Integer, and Floating Point Number box count fields. Many specifications use CONS (or LIST) or the numeric functions from Sections 9, 10, and 11, to construct objects for internal use. For example, many functions use FIX to convert their arguments to Integers, even though in many cases the user cannot actually obtain the Integer constructed (cf. IPLUS in Section 9). In these cases the implementor would naturally be tempted to avoid actually constructing the new object. However, because of the counter fields above, this would be in technical violation of the specifications (since even though the user could not obtain the results of the constructions he could detect whether something was constructed). Since these fields are intended merely to provide the user with a way to monitor his use of these resources, the implementor is hereby encouraged to adopt the more efficient implementations (avoiding the constructions) when possible, despite the technical violation of the specifications.

User calls to VM functions are always associated with frame extensions (simply because such calls are always evaluated using EVAL or APPLY or one of their variants). However, frequently the specifications for VM functions reference other VM functions. Given the conventions on the meaning of $f[x_1;...x_k]$ where f is a VM function (cf. Section 4), no constraints are placed on the implementor regarding how these internal calls to are be

handled. For example, the implementor may choose to implement these calls with the stack mechanism available to the user or to implement them on a private stack used only by internal calls, or to code them "inline".

It is understood that this flexibility with regard to internal control is detectable by the user. For example, calls to STKNTH from within interpreted code will be sensitive to whether or not the recursion of EVAL is visible.

Of course, this private control information is considered part of the "continuation point" of the user process which invoked the VM function evaluation, since, should this process be suspended, the private control information would be necessary in order to resume the process later. Hence, if a private stack for internal control of VM functions is employed, it is considered to be merely a part of the temporaries field of the frame extension of the associated user process.

Because a frame extension is regarded as containing all of the access and control information associated with any function activation (implementations differ only in whether this information is explicitly visible to the user or hidden in the temporaries field), we introduce the following unifying definition:

Definition: A function activation is "controlled from" a frame extension if the frame extension contains the access and control information associated with that function activation.

It is possible for one frame extension to be controlling more than one activation of a VM function. This happens whenever there are several levels of internal calls to VM functions pushed on the private stack within the frame's temporaries field.

We now present the restrictions upon the use of the user's stack by VM functions:

(1) No VM function activation which binds the parameter names of the VM function in a user-visible basic frame may modify the bindings found in that basic frame during the execution of the function body.

(2) A basic frame built to bind the variables (i.e., parameter names or PROG variables) of a VM function may not introduce bindings (of Literal Atoms) which are visible in the access chain from any activation of a user function.

Because user calls to VM functions are always represented on the stack, the error handling facilities can inspect the stack to discover (and possibly diagnose and fix) the cause of the error. One fairly common error handling scenario is as follows: After control has been passed from some VM function to the error package (ERRORX), the package discovers (by inspecting the stack) that the VM function was given the wrong arguments. It decides what the "right" arguments were and modifies the expression being interpreted so that on subsequent encounters with the expression the error should not reoccur. It is desirable to continue the computation at this point, but control cannot be returned to the VM function which caused the error because no assumptions are made about how that function will behave after an error. Thus, the error handler obtains the argument values for the VM function by fetching them from the basic frame of the function (it cannot afford to assume the argument forms can be reevaluated without unintended side-effects) and then applies the VM function to the correct list of values to obtain the result. The error handler then uses RETFROM to jump out of the error and the call to the VM function on the stack, so as to continue the computation without further interruption.

Restriction (1) allows ERRORX and the other user-support facilities (such as DWIM) to use STKARG to recover the initial values of the parameters to the function, thereby permitting the possible diagnosis and recovery from the error.

This restriction has the following implication for the specifications in this document: If a VM function is implemented with some Literal Atom parameter, say X, as the realization of of some meta-variable, say x, in the specification, then X will be bound in the basic frame associated with the activation of the function. If the specification contains a sentence such as: "Let x be x+1", the obvious implementation is to rebind X to the value of (ADD1 X) in the basic frame. This is in violation of restriction (1), since it would destroy the initial binding of X. Instead, the implementor must allot additional storage (either in a lower basic frame or the temporaries field) for the current value of X (and all other modified parameters). We use phrases like "let x be x+1", where x appears as a parameter in the specification, to reduce the number of meta-variable names the reader must wade through.

Restriction (2) prevents variable clashes between VM and user functions. Basically, if a VM function call binds its variables in the usual way and then evaluates user forms, the bindings of the VM function's variables must not conflict the bindings set up for the user. There are two obvious ways to avoid this problem: (1) The contents of the argname fields of the basic frames set up for VM functions can be objects other than Literal Atoms (thereby precluding the possibility of rebinding any user variable) and the code in the body of the function object can reference the argval fields directly (by position rather than argname), or (2) the alink fields of user frames can be set so that no access chain from a user frame includes the basic frame set up for a VM function.

Finally, note that a basic frame is "visible" to the user only if the user is given the opportunity to inspect the stack. In general this may occur during the evaluation of a VM function provided either (1) the VM function itself inspects the stack, (2) the function contains some "safe function calls" (cf. Section 25) permitting interrupts to occur, (3) the function calls EVAL or APPLY on a user supplied form, or (4) the function causes an error. There are many functions for which the first three possibilities do not arise (e.g., LISTP, CONS, etc.). Therefore, these restrictions do not prevent fairly efficient implementations of calls to VM functions, provided appropriate action is taken to update the user stack in the event of an error.

We now consider the implementation of the blip fields. It should be clear that the information contained in the blip fields is necessary for any implementation of the blip-using functions. That is, the fields are really just temporaries of the functions concerned. Should the blip-using functions use the variable binding mechanism supplied to the user, then blip fields are merely the argval fields in basic frames. If blip-using functions use a private access and control mechanism, then the representation of blip fields is entirely up to the implementor. (Of course, even in this case, as part of the private control information for the blip-using function, we consider the blip fields to be in the temporaries field of the frame extension controlling the blip-using function activation.)

To permit the user to inspect and update the contents of the blip fields regardless of the implementation, we provide the functions BLIPSCAN, BLIPVAL, and SETBLIPVAL (specified below). The specifications of these functions rely upon the following definitions.

Definition: A frame extension is said to "contain blip fields" (or "have blip fields") if such fields are associated with the activation of a blip-using function controlled from that frame extension. This definition thus ignores the issue of whether the blip fields are in the basic frame or temporaries field.

Since a frame extension may control more than one VM function activation (in some implementations), a frame extension might contain more than one collection of blip fields.

We must be able to talk conveniently about the ith blip field of a given type from a given frame. We therefore introduce the following definitions.

Definition: The "bliptype blip field sequence of frame", where frame is a frame extension, is the empty sequence if bliptype is not one of the Literal Atoms *FN*, *ARGVAL*, *FORM*, or *TAIL*, or if frame contains no bliptype blip fields. Otherwise, it is the sequence of blip fields obtained by ordering the blip fields of type bliptype in frame in the reverse order of their creation. That is, if frame has n, n > 0, bliptype blip fields, then the 1st element of the bliptype blip field sequence of frame is the newest bliptype blip field in frame (i.e., that which was most recently created), and the nth element is the oldest bliptype blip field in frame.

Definition: The "bliptype blip field sequence in chain", where chain is a chain of frame extensions, is the sequence of blip fields obtained by concatenating (in the order the frames occur in chain) the bliptype blip field sequences of the successive frames in chain.

We now specify the blip processing functions.

```
BLIPSCAN[bliptype;frame]
                Get frame extension frame.
                If there is a frame extension in the clink chain of frame
                which contains a bliptype blip field,
                  create and return a Stack Pointer containing
                  the first such frame extension;
                else, return NIL.

BLIPVAL[bliptype;frame;n]
                Get frame extension frame.
                If n = NIL, let n be 1.

                If n = T:
                  Represent and return as an Integer the number
                  of bliptype blip fields contained in frame;
                else:
                  If not FIXP[n], let n be FIX[n].
                  Let blipseq be the bliptype blip field sequence
                  in the clink chain of frame.
                  If blipseq contains at least n elements,
                    return the contents of the nth blip field in blipseq;
                  else, return NIL.

SETBLIPVAL[bliptype;frame;n;val]
                Get frame extension frame.
                If n = NIL, let n be 1;
                elseif not FIXP[n], let n be FIX[n].

                Let blipseq be the bliptype blip field sequence in
                the clink chain from frame.
                If blipseq contains at leat n elements:
                  Set the contents of the nth blip field in blipseq to val.
                  Return val;
                else, return NIL.
```

20. THE COMPILER

The Virtual Machine does not require the existence of a compiler. However, should one be present, the VM puts certain constraints on it. These are listed below.

The compiler is a function which maps from EXPRs to CEXPRs -- that is, the output of the compiler for a given interpreted function object is a directly executable function object. In some sense, this directly executable function object behaves the same way under evaluation as the original EXPR.

If expr-fnobj is an EXPR function object, and cexpr-fnobj is the output of the compiler for expr-fnobj, then the following conditions must be satisfied:

(1) The implementor can recognize cexpr-fnobj as a function object produced by the compiler.

(2) The parameter n-tuple, eval/noeval type, and spread/nospread type of cexpr-fnobj must be the same as those of expr-fnobj.

(3) The body of cexpr-fnobj may be obtained (by the implementor) and directly executed.

(4) The execution of the body of cexpr-fnobj on any collection of arguments shall cause the same series of function calls (with the same visible effects on the user's stack) as calling expr-fnobj on those arguments, with the following exception: The implementor may designate (and document) a set of so-called "open functions," the code for which may appear "inline" in cexpr-fnobj where calls on these functions appear in expr-fnobj. The code compiled for these open functions may be made more efficient than that executed during interpreted calls by eliminating error checking (provided that incorrect arguments cannot render the state of the VM meaningless), and by eliminating the allocation and maintenance of blip fields. Except for these cases, the code compiled for open functions must cause the same user-visible side-effects (if any) and return the same results as interpreted calls to these functions.

(5) The functions CALLSCCODE and CHANGECCODE must be implementable.

The previous Section makes it clear that the basic frame and frame extension built for a call to a compiled function object is indistinguishable (to the user) from that which would be used for the same call to the interpreted version of that object. Furthermore, given the definition of "call", it is clear that a compiled function can call an interpreted one (and vice versa).

The implementor may wish to provide a range of options for producing more efficient code. This is permitted so long as some arrangement of the options produces code which meets the restrictions above.

Below we introduce definitions which indicate two fairly obvious options. The VM does not require these options of a compiler, however the terms defined below are used in the function CALLSCCODE.

Definition: "(Literal Atom) var is a global variable of (CEXPR) fnobj" if var is a variable of the

EXPR funcion object from which fnobj was produced and selected variable references to var in fnobj have been compiled so as to access the top-level value field of var directly (rather than after a search of the access chain).

In light of this definition, we will refine the notion of "non-local" variables to be those that are not local variables but still involve a search through the access chain.

Definition: A "linked function call" in compiled code is an implementation of the INTERLISP function calling mechanism whereby the function object referenced in the code is that which was in the function's function definition field at the time of the compilation.

Presumably, linked calls are faster since they avoid the reference through the Literal Atom function name and the check to verify that the object is a function object.

```
CALLSCCODE[fnobj;flg]
                If LITATOM[fnobj]. let fnobj be GETD[fnobj].

                If fnobj is a CEXPR:
                  Let localvars be a new proper list of all of the
                  Literal Atoms used as local variables in fnobj.
                  Let nonlocalvars be a new proper list of all of
                  the Literal Atoms used as non-local variables in fnobj.
                  Let globalvars be a new proper list of all of the
                  Literal Atoms used as global variables in fnobj.
                  If flg:
                    Return LIST[NIL:NIL;localvars;nonlocalvars;globalvars];
                  else:
                    Let linkedcalls be a new proper list of all of
                    the functions called with linked calls in the
                    code in fnobj.
                    Let othercalls be a new proper list of all of
                    the functions called without linked function calls
                    in the code in fnobj.
                    Return LIST[linkedcalls;
                                othercalls;
                                localvars;
                                nonlocalvars;
                                globalvars].
```

The following function, CHANGECCODE, destructively replaces all references to one object in some CEXPR by references to another object. We assume the references can be of any nature: named variables, constants, function calls, etc. Note that CHANGECCODE actually modifies the function object rather than copying it.

In order to allow higher level functions to "undo" the effects of the modification we introduce the notion of a "reference map".

Definition: A "reference map for (CEXPR) fnobj" is any object the implementor wishes to use to indicate selected references to objects by the compiled code in fnobj. The function CHANGECCODE constructs and uses references maps.

It is assumed that if a reference map, refmap, gives the locations of (say) all references to some object, x, in fnobj, then after CHANGECCODE is used to replace those references to x

by references to some other object, y, refmap can be validly interpreted as a refence map to selected occurrences of y in fnobj. That is, a reference map does not specify the object referenced, but the references themselves.

```
CHANGECCODE[newref;refmap;fnobj]
                If LITATOM[fnobj], let fnobj be GETD[fnobj].

                If fnobj is a CEXPR:
                  If refmap is not a reference map,
                    let refmap be a new reference map for fnobj giving
                    the locations of all references to the object refmap
                    in fnobj:
                  elseif refmap is not a reference map for fnobj,
                    cause error 17 with culprit
                    CONS["Inconsistent reference map";CONS[refmap;fnobj]].

                  Modify fnobj so that all references indicated in
                  refmap become references to newref.

                  Return refmap.
```

21. FILES AND FILE NAMES

As noted in Section 2, files are not objects, but are assumed to be uniquely identified by file names which are represented as Literal Atoms.

Definition: "the file x" means "the file named x".

Files are used as sources and sinks for input and output functions. These functions are specified in Sections 26 and 27 and deal entirely with transferring sequences of characters to or from files. However, there are some facilities in the VM which involve files but do not cause the transferral of characters. These functions are specified in this Section and embody all the file handling capability of the VM other than mere character transfer.

In many implementations file conventions and file handling facilities are determined by the host operating system, and are beyond the control of the VM LISP implementor. The discussion and specifications in this Section attempt to summarize the assumptions INTERLISP makes about the host filing system. For convenience, we will tag with the phrase "*File Assumption*" any paragraph describing properties of INTERLISP files.

Two distinct kinds of meta-objects are treated as files: input/output devices such as terminals and lineprinters, and secondary storage devices such as discs and drums.

Like strings, files specify a sequence of Charaters. We associate with each character in a file an integer "address" which specifies the number of characters to the left of the addressed character. Thus, the first character in a file has address 0. The characters in a file may be inspected sequentially or (in some cases) randomly.

Convention: The lowest level input operation on a sequentially accessed file will be called "fetching" and returns the next Character in the file. When discussing the analogous operation

for random access files we will specify the position from which the Character is to be fetched. The lowest level output operation on a sequential file will be called "depositing" and transfers a character to the file. When discussing the corresponding random access operation we will specify the target position. None of these operations is available in the VM. We reserve the words "reading" and "writing" for the higher level VM file transfer operations. Reading and writing are specified in terms of these low level operations (cf. Sections 26 and 27).

File Assumption 1: It is assumed the user is directing the computational processes of the VM from an interactive terminal. The assumptions made about the terminal are specified in Section 23.

File Assumption 2: It is assumed that most files will require some initialization before they can be read or written. This will be called "opening" the file and the user must explicitly open a file (except the terminal) before using it. It is assumed that the intended use of a file is declared when it is opened, and this use may be enforced. The uses are declared by supplying "access modes" when the file is opened. These are defined below. Implementations may limit the number of files open simultaneously[11]. Finally, when a file is opened the user may specify the "bytesize" of the file. This is the number of bits which must be fetched or deposited to represent one Character on the file. In general the VM supports only one bytesize: the standard VM bytesize. When a VM function is called upon to fetch from or deposit to a file with a non-standard bytesize, the implementor is free to truncate or pad the character codes as necessary. The implementor is also free to extend the VM facilities for reading from such files, or to provide additional fetching and depositing facilities.

Definition: An "access mode" is one of the Literal Atoms INPUT, OUTPUT, BOTH, or APPEND. The relationship between the access mode used when a file is opened and subsequent use of the file is specified below:

Access Mode	Use
INPUT	The file may be read from only. When the file is opened its file pointer field must be set to 0 (see below).
OUTPUT	The file may be written to only. When the file is opened its file pointer field must be set to 0.
BOTH	The file may be read from and written to. When opened the file pointer field must be set to 0.
APPEND	The file may be written to only. When the file is opened its file pointer field must be set to the contents of the end of file pointer field (see below).

The VM associates three fields with every open file. The names of these fields are:

(1) position field,
(2) file pointer field,
(3) end of file pointer field.

---

11 INTERLISP-10 restricts it to 16.

Each field contains an integer (note lower case). The contents of these fields are specified below.

The intuitive purpose of the position field is to maintain an indication of how many characters have been deposited to (or fetched from) the file since the last carriage return character was written (fetched). The precise manipulation of this field is left to the implementor. The least sophisticated procedure is outlined below:

(1) Whenever a file is opened, its position field is set to 0.

(2) Whenever any non-carriage return character is fetched from a file the contents of the position field of that file is incremented by 1 and stored back into the position field.

(3) Whenever a carriage return character is fetched from a file the position field of that file is set to 0.

(4) Whenever any non-carriage return character is deposited in a file, the contents of the position field of that file is incremented by 1 and stored back into the position field.

(5) Whenever the carriage return character is deposited in any file, the position field of that file is set to 0.

The implementor may choose to implement (and document) elaborations on this procedure. For example, the procedure might be sensitive to the primary Terminal Table (see Section 24) when dealing with the terminal since some characters (such as <tab> or <form-feed>) might require more than one character position to print.

The intuitive purpose of the file pointer field is to specify the target address from which (or to which) a character is to be fetched (deposited). The initial contents of a file's pointer field is determined according to how the file is opened and is specified in the function OPENFILE. The definitions of reading and writing characters specify the actual use and manipulation of the file pointer field.

The end of file pointer field always contains the number of characters in the file. When a file is opened, this number is computed and stored in the end of file pointer field for the file. (The end of file pointer field is unspecified for the terminal.) The next file assumption specifies how the end of file pointer is maintained.

File Assumption 3: If a character is deposited in a file at an address, n, which is less than the end of file pointer, eof, the old character at address n is overwritten with the new character. If n is equal to or greater than eof, the file is expanded by the addition of n-eof+1 unspecified characters to the right of the current last character, the end of file pointer is set to n+1, and the character is then deposited at address n.

File Assumption 4: When operations on a file are complete, it is assumed some terminating actions may be performed. This is called "closing" the file and usually files are explicitly closed by the user.

File Assumption 5: If a non-existent file with an acceptable name is opened for output the effect is the same as though an existing file, having the same name and containing no characters, had been opened.

Convention: A file is said to permit "random access" if it is possible for the user to set the file's file pointer.

File Assumption 6: The user can create, read, and write stored files permiting random access.

It is recognized that some file systems allow certain abbreviations and default conventions when specifying file names in various contexts. Thus we distinguish two kinds of file names, those that contain abbreviations or rely upon defaults supplied by the underlying file system, and those that represent the fully specified file name.

Convention: A "full file name" is a character sequence not depending on abbreviations or defaults to specify a unique file.

For convenience we allow character sequences which do not represent full file names to be "recognized" as abbreviations for full names, should the host filing system or implementor choose to supply such a scheme. We allow such an abbreviation to actually denote one of several files and introduce the notion of a "recognition mode" to distinguish precisely one of the possible matches.

Definition: A "recognition mode" is one of the Literal Atoms OLD, NEW, or OLDEST. The mode places certain restrictions on the file denoted by a recognized name. These restrictions are given below.

Definition: "(character sequence) name is recognized in (recognition mode) mode" if the filing system's naming conventions allow name to denote a unique file, file, with full file name, fullname, satisfying the property required by mode. The mode properties are:

mode	property
OLD	file is the most recently created existing file which name could denote.
NEW	file does not yet exist, but the user could create a new file with full name fullname.
OLDEST	file is the oldest existing file which name could denote.

File Assumption 7: INTERLISP assumes that both full file names and those that can be recognized are the names of Literal Atoms (i.e., neither denotes an INTERLISP Number).

It is actually the case that the high level facilities in INTERLISP make some assumptions about the form and characteristics of file names themselves. For example, in INTERLISP-10, which relies on the file naming conventions of TENEX, functions which create new files "know" that file names have extensions and version numbers appended to the end of the "main name" and separated by the characters '.' and ';' respectively. The VM does not require these naming conventions, but it is probable that that part of the high level code which generates file names will have to be reimplemented to suit the local conventions. This was deemed more practical than trying to standardize file name conventions.

Definition: A "File Name" (note capitalization) is a Literal Atom whose name is a file name. We will use the adjectives "recognizable" and "full" in the obvious way.

The File Name T is reserved for the interactive terminal the user is presumed to be using.

The file T is always open for both input and output. The implementor is free to supply an arbitrary number of reserved recognizable File Names for online site-dependent devices such as lineprinters, etc.

The VM also provides a facility for producing "typescript" files, that is, files containing all of the input/output transactions with the terminal. The user may designate one file to be used for this purpose (see DRIBBLE below). Sections 26 and 27 specify what is written to this file.

INTERLISP maintains three distinguished full file names (and thus, three distinguished files). These file names are the default file names for input and output (i.e., they are used when any VM file handling function is given NIL instead of a File Name) and the name of the current typescript file (if any). The corresponding files are called the "primary input file", the "primary output file", and the "dribble file". Initially, the first two are T and there is no dribble file.

```
FULLNAME[litatom;recog]
                If not LITATOM[litatom], cause error 14 with culprit litatom;
                elseif recog is NIL, let recog be OLD;
                elseif recog is not a recognition mode,
                  cause error 27 with culprit recog.

                If litatom is recognized in recognition mode recog as an
                abbreviation for some file with full name fullname,
                  return fullname;
                else, return NIL.

OPENFILE[file;access;recog;bytesize]
                Let fullname be FULLNAME[file;recog].
                If fullname is NIL, return NIL.
                If access is not an access mode,
                  cause error 27 with culprit access.
                If bytesize is NIL,
                  let bytesize be the standard VM bytesize;
                else, let bytesize be FIX[bytesize].
                If the implementation defined limit on the
                number of open files has been reached,
                  cause error 15 with culprit NIL.

                Open file fullname with access access and byte size
                bytesize, and should it be found impossible to
                do so (e.g., due to a protection violation)
                cause error 9 with culprit fullname.
                Return fullname.

OPENP[file;access;recog]
                If file is not a Literal Atom,
                  cause error 27 with culprit file;
                elseif file is NIL:
                  If access is NIL:
                    Create and return a proper list of the full File Names
                    of all open files (excluding T and the dribble file,
                    if any);
                  elseif access is an access mode:
                    Create and return a proper list of the full File Names
                    of all files open for the mode of access specified
                    by access (excluding T and the dribble file, if any);
                else (file is a non-NIL Literal Atom):
                  If access is NIL, let access be INPUT.
                  If recog is NIL:
```

```
                    If access is OUTPUT, let recog be NEW;
                    else, let recog be OLD.
                  Let fullname be FULLNAME[file;recog].
                  If fullname is open for the mode of access specified
                  by access, return fullname;
                  else, return NIL.

INPUT[file]       If file is NIL, return the full File Name of the
                  current primary input file.
                  Let fullname be OPENP[file;INPUT].
                  If fullname is NIL, cause error 13 with culprit file.
                  Let oldfile be the full File Name of the current
                  primary input file.
                  Set the primary input file to fullname.
                  Return oldfile.

INFILE[file]      Return INPUT[OPENFILE[file;INPUT;OLD]].

INFILEP[file]     Return FULLNAME[file;OLD].

OUTPUT[file]      If file is NIL, return the full File Name of
                  the current primary output file.
                  Let fullname be OPENP[file;OUTPUT].
                  If fullname is NIL, cause error 13 with culprit file.
                  Let oldfile be the full File Name of the current
                  primary output file.
                  Set the primary output file to fullname.
                  Return oldfile.

OUTFILE[file]     Return OUTPUT[OPENFILE[file;OUTPUT;NEW]].

OUTFILEP[file]    Return FULLNAME[file;NEW].

IOFILE[file]      OPENFILE[file;BOTH;OLD].

DRIBBLE[file]     If file=T, let file be NIL.
                  If there is a dribble file:
                    Let oldfile be the full File Name of the dribble file.
                    If oldfile is the primary output file,
                      set the primary output file to T.
                    Close oldfile;
                  else, let oldfile be NIL.

                  If file /= NIL:
                    If OPENP[file;OUTPUT], let newfile be OPENP[file;OUTPUT];
                    else, let newfile be OPENFILE[file;OUTPUT;NEW].
                    Set the dribble file to newfile.

                  Return oldfile.

DRIBBLEFILE[]     If there is a dribble file,
                    return the full File Name of the current dribble file;
                  else, return NIL.

CLOSEF[file]      If file is NIL:
                    If the primary input File Name is not T,
                      let fullname be the primary input full File Name;
                    elseif the primary output File Name is not T,
                      let fullname be the primary output full File Name;
                    else return NIL;
```

```
                else:
                  Let fullname be OPENP[file].
                  If fullname is NIL, cause error 13 with culprit file.
                If fullname is the primary input File Name,
                  set the primary input file to T.
                If fullname is the primary output File Name,
                  set the primary output file to T.
                If there is a dribble file and it is fullname:
                  Return NIL;
                else:
                  Close file fullname.
                  Return fullname.
```

Note: The dribble file cannot be closed with CLOSEF.

```
CLOSEALL[]      Let lst be a new proper list of the full names of all open
                files (except T and the dribble file, if any).
                For each filename in lst do:
                  CLOSEF[filename];
                Return lst.

RANDACCESSP[file]
                If file is NIL, let file be the primary input file;
                elseif OPENP[file], let file be OPENP[file];
                else, cause error 13 with culprit file.
                If file permits random access, return file;
                else, return NIL.

GETFILEPTR[file]
                If file is NIL, let file be the primary input file;
                elseif OPENP[file], let file be OPENP[file];
                else, cause error 13 with culprit file.
                Represent and return as an Integer the contents
                of the file pointer field of file.

GETEOFPTR[file] If file is NIL, let file be the primary input file;
                elseif OPENP[file], let file be OPENP[file];
                else cause error 13 with culprit file.
                Represent and return as an Integer the contents
                of the end of file pointer field of file.

SETFILEPTR[file;val]
                If RANDACCESSP[file], let file be RANDACCESSP[file];
                else, cause error 17 with culprit file.
                If not FIXP[val], let val be FIX[val].
                If val < -1, cause error 27 with culprit val;
                elseif val=-1:
                  Set the file pointer field of file to the
                  contents of the end of file pointer field of file.
                  Return val;
                else:
                  Set the file pointer field of file to the integer
                  represented by val.
                  Return val.

POSITION[file;val]
                If file is NIL, let file be the primary output file;
                elseif OPENP[file], let file be OPENP[file];
                else, cause error 13 with culprit file.
```

```
                Let oldval be the Integer representing the
                contents of the position field of the file file.
                If val is NIL, return oldval.
                If not FIXP[val], let val be FIX[val].
                Set the position field of file to the integer
                represented by val.
                Return oldval.

DELFILE[file]   Let fullname be FULLNAME[file;OLDEST].
                If fullname is not NIL:
                  If fullname is open, cause error 17 with culprit
                  CONS["Close file before deleting";fullname].
                  Delete file fullname.
                  Return fullname;
                else, cause error 23 with culprit file.

RENAMEFILE[file;newname]
                Let file be INFILEP[file].
                If file is NIL, return NIL.
                If file is open, cause error 17 with culprit
                CONS["Close file before renaming";file].
                Let newname be OUTFILEP[newname].
                If newname is an existing file, return NIL.
                Rename the file file to have name newname.
                Return newname.
```

22. READ TABLES

Read Tables are objects that specify the syntactic properties of characters for the input (and some output) routines. Since the input routines are concerned with parsing incoming character sequences into objects, the Read Table in use at the time determines which sequences are recognized as Literal Atoms, List Structures, etc.

We will present the specifications of the input/output functions in Sections 26 and 27. This Section is concerned with the manipulation of the Read Tables themselves.

Each character must belong to precisely one "syntax class". By definition of a syntax class, all characters in a given syntax class exhibit identical syntactic properties. There are nine basic syntax classes, each associated with a primitive syntactic property, and then an unlimited assortment of user-defined syntax classes (jointly referred to as "read macros" but individually constituting unique syntax classes).

For example, the characters which indicate the beginning of (a character sequence representing) a List Structure form a basic syntax class. The general property uniting all read macro characters is that a user-specified computation is performed to determine the syntactic effect of each character.

It should be noted that a "syntax class" is an abstraction provided by the VM. There is no object referencing a collection of characters and called a Syntax Class. A Read Table provides the association between a character and its syntax class, and the input/output routines enforce the abstraction by using Read Tables to drive the parsing.

To allow the user to specify the association between Characters and syntax classes we must introduce names for the basic syntax classes and the attributes of read macros.

Definition: A "basic syntax class" is one of the Literal Atoms LEFTPAREN, RIGHTPAREN, LEFTBRACKET, RIGHTBRACKET, STRINGDELIM, ESCAPE, BREAKCHAR, SEPRCHAR, and OTHER.

The properties of these classes are defined in Sections 26 and 27. Briefly, the first four classes mark character sequences representing List Structures, STRINGDELIM marks Strings, BREAKCHAR and SEPRCHAR mark Literal Atoms and Numbers, ESCAPE provides a mechanism for inputting these syntactically special characters, and OTHER is the class of all other characters except those that are read macros.

It is convenient to refer to some of these clases jointly.

Definition: The "break syntax classes" are LEFTPAREN, LEFTBRACKET, RIGHTPAREN, RIGHTBRACKET, STRINGDELIM, and BREAKCHAR.

The syntactic properties of read macros are determined by the vaues of five attributes.

Definition: The five read macro attributes are "type", "context", "wakeup mode" "escape flag", and "body". Each attribute may take on one of the discrete "legal" values shown below:

attribute	legal value
type	MACRO, SPLICE, INFIX
context	ALWAYS, FIRST, ALONE
wakeup mode	WAKEUP, NOWAKEUP
escape flag	ESCQUOTE, NOESCQUOTE
body	a Literal Atom or function object

Briefly, the meanings of these attributes are as follows: The body specifies a computation to be performed when the character is read in a certain syntactic context specified by the context attribute. The type attribute determines what is done with the value of the computation. The wakeup mode attribute is important only when the macro is read from the terminal and, if it is WAKEUP, means that the reading routines should begin reading and parsing the characters in the line buffer as soon as the read macro character has been deposited in that buffer (see Sections 23 and 27). The escape flag attribute affects how the character is to be printed.

It is convenient to define a read macro specification itself as a 5-tuple meta-object, containing INTERLISP objects. We use the notion of a meta-object because the user specifies the individual components but does not supply a read macro specification as an object.

Definition: A "read macro specification" is a 5-tuple meta-object containing INTERLISP objects: <type, context, wakeup mode, escape flag, body> where the components are legal values of the corresponding read macro attributes.

Definition: A "syntax class specification" is either one of the basic syntax classes or a 5-tuple read macro specification.

It is sometimes desirable to prevent all read macros in a Read Table from invoking computation, even though their attributes would otherwise allow it. When in such a state, we say that the read macros are "disabled". This is controlled by a flag field in the Read Table.

Definition: A "Read Table" is an object with the following properties:

(1) For each character, c, there is a field which contains a syntax class specification, called the "syntax class of c".

(2) There is a binary field, called the "read macros enabled" field, which may contain T or NIL.

Read Tables constitute a distinct class of objects with class name READTABLEP.

Definition: "(character) char is a LEFTPAREN of (Read Table) rdtbl", if the contents of the char syntax class field of rdtbl contains LEFTPAREN. Analogous definitions are asserted for the other syntax classes. A "break character of rdtbl" is any character having one of the *break syntax classes in its syntax class field of* rdtbl. A "*separator*" character is one having SEPRCHAR in its syntax class field of rdtbl.

Because the user is allowed to specify the syntax class to which each character belongs, we must define the process by which the implementor translates user-supplied objects describing read macros into (meta-object) read macro specifications.

Definition: To "obtain the 5-tuple corresponding to (proper list) lst (with length n)" means:

```
"If n<2, cause error 27 with culprit lst.
If CAR[lst] is a legal type, type,
  let the type of the 5-tuple be type;
else cause error 27 with culprit lst.
If the last element of lst is a legal body, body,
  let the body of the 5-tuple be body.
else cause error 27 with culprit lst.

Consider the remaining n-2 elements in lst as a set, remainder,
but using ESCQUOTE in place of all occurrences of ESC
and NOESCQUOTE in place of all occurrences of NOESC.

If there is more than one legal context (or wakeup mode or escape flag)
attribute value in remainder,
  cause error 27 with culprit lst.
If any element of remainder is not a legal context or wakeup mode or
escape flag value,
  cause error 27 with culprit lst.
If there is no context value in remainder,
  add ALWAYS to remainder (i.e., let remainder be the new set
  obtained by adding the element ALWAYS to remainder).
If there is no wakeup mode value in remainder,
  add NOWAKEUP to remainder.
If there is no escape flag value in remainder,
  add ESCQUOTE to remainder.
Let the context, wakeup mode, and escape flag components
in the 5-tuple be the context, wakeup mode, and escape flag
values in remainder."
```

Similarly, we must define the process used by the implementor to construct an object (for the user) which contains all of the information in a read macro specification.

Definition: To "create a proper list corresponding to (a read macro specification) <type, context, wakeup, escape, body>" means "LIST[type;context;wakeup;escape;body]".

Thus, the proper list the user supplies to a Read Table to define a read macro is not the same (EQ) proper list the user obtains when querrying the Read Table. In fact, the two proper lists may not even be the same length or contain the same elements. However, both translate into the same 5-tuple.

The VM maintains three distinguished Read Tables. The first is called the "original" Read Table. This Read Table may not be obtained by the user and is used to provide a way to recover the initial settings in the other two distinguished Read Tables. The second is called the "system" Read Table, and is the one used when the system itself is interacting with the user's terminal (e.g., reading for the top-level input to EVALQT). The third is called the "primary" Read Table, and is the default Read Table for user programs. The latter two Read Tables initially contain copies of the original Read Table (i.e., they are distinct Read Table objects containing the same settings). We will be precise regarding the use of these three tables when precision is required. For the moment it is sufficient simply to state the existence of these three Read Tables.

Provided the characters are available in the implementation, the following associations should be found in the original Read Table:

Character	Initial Syntax Class Specification
<tab>	SEPRCHAR
<carriage return>	SEPRCHAR
<linefeed>	SEPRCHAR
<formfeed>	SEPRCHAR
<end-of-line>	SEPRCHAR
<blank>	SEPRCHAR
"	STRINGDELIM
<percent>	ESCAPE
(	LEFTPAREN
)	RIGHTPAREN
[	LEFTBRACKET
]	RIGHTBRACKET

All other characters should have syntax class OTHER. The read macros enabled field of the original Read Table is set to T.

```
READTABLEP[x]   If x is a Read Table, return x;
                else, return NIL.

GETREADTABLE[rdtbl]
                If rdtbl is NIL, return the primary Read Table;
                elseif rdtbl is T, return the system's Read Table;
                elseif READTABLEP[rdtbl], return rdtbl;
                else, cause error 38 with culprit rdtbl.

SETREADTABLE[rdtbl;flg]
                let rdtbl be GETREADTABLE[rdtbl].
                If flg:
                  Let oldrdtbl be the system's Read Table.
                  Set the system's Read Table to rdtbl.
                  Return oldrdtbl.
                else:
                  Let oldrdtbl be the primary Read Table.
                  Set the primary Read Table to rdtbl.
                  Return oldrdtbl.
```

```
RESETREADTABLE[rdtbl;source]
                If not READTABLEP[rdtbl],
                  let rdtbl be GETREADTABLE[rdtbl].
                If not READTABLEP[source]:
                  If source = ORIG,
                    let source be the orignal Read Table;
                  else, let source be GETREADTABLE[source].

                If the read macros enabled field of source is T,
                  set the read macros enabled field of rdtbl to T;
                else, set the read macros enbabled field of rdtbl to NIL.

                For each character, c, replace the c syntax class field in
                rdtbl with the contents of the c syntax class field in source.

                Return rdtbl.

COPYREADTABLE[rdtbl]
                Let newrdtbl be a new Read Table.
                Return RESETREADTABLE[newrdtbl;rdtbl].
```

The following two functions operate on both Read Tables and Terminal Tables. Terminal Tables are described in Section 24. One of their functions is to specify "terminal syntax classes" for characters. For details and definitions of terms used, see Section 24.

```
GETSYNTAX[char;tbl]
                Let origchar be char.
                If char is a Character or character code
                (treat Characters 0 through 9 as character codes),
                  let char be the corresponding character.

                If not READTABLEP[tbl] and not TERMTABLEP[tbl]:
                  If char is a terminal syntax class:
                    If tbl = ORIG,
                      let tbl be the original Terminal Table;
                    else, let tbl be GETTERMTABLE[tbl];
                  elseif tbl = ORIG,
                    let tbl be the original Read Table;
                  else, let tbl be GETREADTABLE[tbl];

                If char is a character:
                  If READTABLEP[tbl]:
                    Let class be the contents of the char syntax class
                    field in tbl.
                    If class is a 5-tuple,
                      create and return a proper list corresponding to class;
                    else return the basic syntax class class;
                  else (tbl in a Terminal Table):
                    Return the contents of the terminal syntax
                    class field of char in tbl;
                elseif char is a basic syntax class:
                  If TERMTABLEP[tbl], cause error 38 with culprit tbl.
                  Create and return a proper list of all of the
                  character codes whose syntax class fields in tbl
                  contain char.
                elseif char = BREAK:
                  If TERMTABLEP[tbl], cause error 38 with culprit tbl.
                  Create and return a new proper list of all of the
                  character codes whose syntax class fields in
                  tbl contain one of the break syntax classes.
```

```
                elseif char is a terminal syntax class:
                  If READTABLEP[tbl], cause error 39 with culprit tbl.
                  Create and return a proper list of all of the character
                  codes whose terminal syntax class fields in tbl
                  contain char;
                else, cause error 27 with culprit origchar.

SETSYNTAX[char;class;tbl]
                Let origchar be char.
                If char is a Character or character code (treat
                Characters 0 through 9 as character codes),
                  let char be the corresponding character;
                else, cause error 27 with culprit char.

                If not READTABLEP[tbl] and not TERMTABLEP[tbl]:
                  If TERMTABLEP[class] or class is a Terminal Syntax Class,
                  let tbl be GETTERMTABLE[tbl];
                  else, let tbl be GETREADTABLE[tbl];

                If READTABLEP[class] or TERMTABLEP[class]
                or class = NIL or class = T or class = ORIG:
                  Let class be GETSYNTAX[charcode;class],
                  where charcode is the character code corresponding to char;
                elseif class is a Character or character code:
                  Let class be GETSYNTAX[class;tbl];
                elseif class = SEPR:
                  Let class be SEPRCHAR.
                Let oldclass be GETSYNTAX[char;tbl].

                If class is a basic syntax class:
                  If TERMTABLEP[tbl], cause error 38 with culprit tbl.
                  Set the char syntax class field in tbl to class.
                  Return oldclass;
                elseif class = BREAK:
                  If TERMTABLEP[tbl], cause error 38 with culprit tbl.
                  If oldclass is not a break syntax class or a read
                  macro specification, set the char syntax class field
                  in tbl to BREAKCHAR.
                  Return oldclass.
                elseif LISTP[class]:
                  If TERMTABLEP[tbl], cause error 38 with culprit tbl.
                  Obtain the 5-tuple corresponding to class, and
                  set the char syntax class field in tbl to this 5-tuple.
                  Return oldclass.
                elseif class is a terminal syntax class name:
                  If READTABLEP[tbl], cause error 39 with culprit tbl.
                  If class = NONE, set the char terminal syntax class
                  field in tbl to NONE;
                  elseif char is not a special terminal
                  character, cause error 27 with culprit origchar.
                  else:
                    If there is any character, c, whose
                    terminal syntax class field in tbl contains class,
                    set the c terminal syntax class field in tbl to NONE.
                    Set the char terminal syntax class field
                    in tbl to class.
                    Return oldclass;
                else, cause error 27 with culprit class.

GETBRK[rdtbl]   Return GETSYNTAX[BREAK;rdtbl].
```

```
GETSEPR[rdtbl]  Return GETSYNTAX[SEPRCHAR;rdtbl]

SETBRK[lst;flg;rdtbl]
                If lst=T:
                  If rdtbl=T, let lst be GETBRK[ORIG];
                  else, let lst be GETBRK[T].

                (We assume lst is a proper list of either
                Characters or character codes.)

                If flg=NIL:
                  For every element, char, of GETBRK[rdtbl] do:
                    SETSYNTAX[char;OTHER;rdtbl].
                  For every element, char, of lst, do:
                    SETSYNTAX[char;BREAKCHAR];
                elseif flg=0:
                  For every element, char, in lst do:
                    SETSYNTAX[char;OTHER;rdtbl];
                elseif flg=1:
                  For every element, char, in lst do:
                    SETSYNTAX[char;BREAKCHAR;rdtbl].

                Return NIL.

SETSEPR[lst;flg;rdtbl]
                (Same specification as for GETBRK except
                use GETSEPR for GETBRK and SEPRCHAR for BREAKCHAR.)

READMACROS[flg;rdtbl]
                If not READTABLEP[rdtbl],
                  let rdtbl be GETREADTABLE[rdtbl].
                Let oldflg be the contents of the
                read macros enabled field of rdtbl.
                If flg, set the read macros enabled field of rdtbl to T;
                else, set the read macros enabled field of rdtbl to NIL.
                Return oldflg.
```

23. TERMINALS

This Section describes the assumptions the VM makes about the terminal input/output capabilities of the underlying operating system or machine.

As *File Assumption* 1 makes clear, the VM assumes the user is directing the computations of the VM from an interactive terminal. The VM allows the implementor to class terminals as either "display" or "non-display". The only distinction made by INTERLISP is that if the user's terminal is a display terminal some of the high-level facilities assume that information can be displayed to the user faster than with a non-display terminal, and hence (in their default mode) supply more information.

The next assumption about the terminal concerns interrupt characters.

Terminal Assumption 1: Whenever certain characters (determined under software control) are typed at the terminal, an implementor supplied procedure is immediately invoked, regardless of any ongoing computational processes. If a character causes such an invocation, it is called an "interrupt character". Section 25 deals with the VM interrupt facilities.

We next introduce the concept of a "buffer" and define the operations of "fetching" and "depositing" on buffers. Intuitively, a buffer is just a queue of characters.

Definition: A "buffer" of length n>0 is a meta-object with the following properties:

(1) There are n character fields, each identified by an integer, 1=<i=<n.

(2) There is a field, called the "deposit pointer" field, which contains a non-negative integer not exceeding n.

When a buffer is created, its deposit pointer field is set to 0.

Definition: If the deposit pointer of a buffer is 0, the buffer is said to be "empty". To "clear" a buffer is to set its deposit pointer to 0. If the deposit pointer is equal to the length of the buffer, the buffer is said to be "replete". We reserve the word "full" for future use (cf. Section 27).

Definition: If a buffer, buff, is non-empty, then to "fetch" the next character from it means:

```
"Let c be the character in the first field of buff.
Let n be the deposit pointer of buff.
For every integer i, 2=<i=<n, set the contents of
the i-1st field of buff to the contents of the ith field of buff
(i.e., shift all of the characters in buff to the left by 1).
Set the deposit pointer of buff to n-1.
Return c."
```

Definition: If a buffer, buff, is not replete, then to "deposit" a character, c, in the buffer means:

```
"Let n be the contents of the deposit pointer field of buff.
Set the deposit pointer field of buff to n+1.
Set the n+1st character field of buff to c."
```

Of course, buffers need not be implemented this way as long as the functional behavior of depositing and fetching is preserved.

At any instant a buffer can be considered to correspond to the character sequence which would be obtained by fetching successive characters from the buffer until it was empty.

Terminal Assumption 2: There is a buffer of unspecified length, called the "system input buffer", with the property that whenever any character, char, is typed at the terminal, the following procedure is followed:

```
"If char is the CTRLV character of the primary
Terminal Table (cf. Section 24):
  Let char be the next character typed at the terminal.
  If the system input buffer is replete:
    Send the bell character to the terminal (i.e., ^G in ASCII,
    or, if unavailable, some character which when sent to the
    terminal will alert the user that typein is being ignored).
    Ignore char.
  else, deposit char in the system input buffer;
elseif char is a valid interrupt character (cf. Section 25) and the
interrupt class of char, class, is something other than NONE:
  (Note:  The terms used in this clause are defined in Section 25.)
  If the interrupts armed field contains T, immediately
```

```
    perform the computation specified for the class interrupt;
  else, set the saved interrupt character field to char.
  Ignore char (i.e., do not deposit it in the system input buffer).
else:
  If the system input buffer is replete:
    send the bell character to the terminal (and ignore char):
  else, deposit char in the system input buffer."
```

Terminal Assumption 3: Whenever a fetch is requested from the system input buffer while that buffer is empty, all of the user's computational processes are halted until the user begins typing at the terminal.

Recognizing that the VM is often implemented in a time shared environment, the following assumption is made.

Terminal Assumption 4: There is a buffer of unspecified length, called the "system output buffer", with the property that whenever any character is "deposited in the terminal" it is actually deposited into this buffer. It is assumed a concurrent process is actually fetching characters from this buffer and transferring them to the actual terminal.

In the next Section we outline the distinguishing characteristics of terminal input/output and present the data structure which specifies how the input/output routines should behave with respect to the terminal.

```
DISPLAYTERMP[]  If the terminal is a display terminal, return T;
                else, return NIL.

DOBE[]          Wait until the system output buffer
                is empty and then return NIL.
```

Note: "DOBE" stands for "dismiss until output buffer empty".

24. TERMINAL TABLES

Terminal Tables are objects which supply the input/output routines with information specifically pertaining to the file T. Because the terminal is an interactive source/sink it has characteristics not found in any other file.

The following special characteristics are recognized for terminal input/output:

(1) Some characters should cause interrupts as soon as they are typed (to allow the termination of infinite computations, for example).

(2) Some subset of the characters may be reserved for editing type-in during input by the user.

(3) Many control characters in the alphabet do not usually perform meaningful functions when deposited in the terminal and there should therefore be special provisions for outputting them.

(4) It is usually necessary to echo (i.e., print to the terminal) characters read from the terminal.

(5) It is sometimes necessary to perform lower to upper case conversion on characters read from the terminal.

These characteristics suggest a variety of facilities that are offered by the VM input/output routines when dealing with the terminal. All of these facilities except those suggested by (1) are controlled by Terminal Tables.

In this Section we will deal with Terminal Tables as data objects. Sections 26 and 27 will actually specify how they interact with the input/output routines. Section 25 deals with the facilities suggested by (1).

As usual, we will present a brief discussion of the features controlled by Terminal Tables, simply to motivate the contents of the various fields involved. We start with the editing of characters, as suggested by (2).

To permit interactive editing of typed input, characters are fetched one at a time from the system input buffer and deposited in an internal VM buffer (called the "line buffer") where they are subject to editing until a "wakeup" character is read. The wakeup character constitutes one of several "terminal syntax classes". There are four editing operations which can be performed on characters in the buffer, and each operation may be triggered by no more than one character. Each operation has a name and defines a distinct terminal syntax class of the same name. We then add a sixth terminal syntax class which contains all the rest of the characters.

Definition: A "terminal syntax class" is one of the Literal Atoms WAKEUPCHAR, CHARDELETE, LINEDELETE, RETYPE, CTRLV, or NONE.

We will briefly (and informally) explain the editing operations. Their formal definitions are in Section 27. The first five classes can contain at most one character each. The WAKEUPCHAR character breaks the incoming stream into segments to be edited independently. The CHARDELETE character causes the deletion of the last non-editing character read. The LINEDELETE character causes the entire line buffer to be cleared. The RETYPE character causes the line buffer to be printed to the terminal for inspection. The CTRLV character provides a mechanism for entering control characters into the sequence (even those with editing or interrupt functions). Finally, NONE is the class of all remaining characters. (NONE is used instead of OTHER so that GETSYNTAX and SETSYNTAX can distinguish a Read Table syntax class from a Terminal Table syntax class.)

Recognizing that some operating systems have flexible terminal editing and control facilities based on a preferred subset of characters, it is permissable to limit the WAKEUPCHAR, CHARDELETE, LINEDELETE, RETYPE, and CTRLV characters to those preferred by the host system.

Definition: A "special terminal character" is a character permitted by the implementation to be in the terminal syntax classes WAKEUPCHAR, CHARDELETE, LINEDELETE, RETYPE, and CTRLV. The set of special terminal characters must contain at least 5 distinct characters and must be documented by the implementor

The output protocol during the character and line deletion operations in the line buffer can be specified by the user. There are five "messages" (character sequences) associated with these two operations.

Definition: A "deletion control message name" is one of the Literal Atoms LINEDELETE, 1STCHDEL, NTHCHDEL, POSTCHDEL, and EMPTYCHDEL.

Briefly, the five respective messages are printed when the LINEDELETE character is read, when the first of a series of CHARDELETE characters is read, when the nth consecutive CHARDELETE character is read, when the first non-CHARDELETE non-editing character is read after a CHARDELETE, and when a CHARDELETE character is read when there are no characters in the buffer.

It is also possible to specify whether or not the characters deleted by CHARDELETE are echoed when deleted.

Finally, there is a mechanism which allows the user to defeat the line buffering.

This concludes the survey of facilities suggested by characteristic (2) of terminal input/output. Section 27 presents the details. Next, we consider the problem of non-printing control and formatting characters.

It does not make a great deal of sense to deposit most control and formatting characters to the terminal. This is either because the functions traditionally performed by such characters are not meaningful in an user-controlled interactive environment (e.g., end-of-transmission), or because the necessary hardware formatting capability is not present in the terminal (e.g., form-feed). It is therefore useful to provide a range of "control character echo modes" for each control character (independently). These modes specify different ways of dealing with the problem of echoing or writing control characters to terminals.

Definition: A "control character echo mode" is one of the Literal Atoms IGNORE, REAL, SIMULATE, or UPARROW.

If a control character has echo mode IGNORE, then it is simply not deposited in the terminal. If the mode is REAL, the control character is deposited and the terminal hardware is expected to deal with it. If the mode is SIMULATE then (when possible) a sequence of characters will be deposited which simulate the effect of the character (e.g., a simulated tab will deposit a sequence of spaces). Finally, if the mode is UPARROW the character is printed as the '↑' character followed by the control character's *equivalent. The details are presented in* Section 26.

Characteristics (4) and (5) of terminal input/output imply that the user should exercise control over whether any characters are echoed, and whether they are converted to upper case. The facilities for these features are specified in Section 27.

We are now in a position to state the characteristics of a Terminal Table.

Definition: A "Terminal Table" is an object with the following properties:

(1) For each character there is a field containing a terminal syntax class, with the restrictions that at most one character may have CHARDELETE (or LINEDELETE or RETYPE or CTRLV) in its field and only special terminal characters may have CHARDELETE, LINEDELETE, RETYPE, CTRLV, or WAKEUPCHAR *in their syntax class fields.*

(2) For each deletion control message name, there is a field containing a (meta-object) character sequence (possibly limited to an unspecified number of

characters[12]).

(3) There is a binary field, called the "control" field, containing either T or NIL (determining whether line buffering is performed).

(4) For each control character there is a field containing a control character echo mode.

(5) There is a binary field, called the "deleted character echo mode" field, which contains either ECHO or NOECHO.

(6) There is a binary field, called the "global echo mode" field, which contains either T or NIL.

(7) There is a ternary field, called the "lower-to-upper case conversion mode" field, containing either T, 0, or NIL.

Terminal Tables constitute a distinct class of objects with class name TERMTABLEP.

Definition: "(character) x is the (or a) CHARDELETE character of (Terminal Table) y" if the x terminal syntax class field of y contains CHARDELETE. Analogous definitions are asserted for the other terminal syntax classes.

There are two distinguished Terminal Tables, called the "original" Terminal Table and the "primary" Terminal Table. The original Terminal Table is analogous to the original Read Table. The primary Terminal Table is initially set to a copy of the original Terminal Table and is used when any input/output operation uses the file T (which is the only time any Terminal Table is ever used).

Initially, the original Terminal Table shall contain the following settings, assuming these characters are available as special terminal characters:

character	terminal syntax class
↑A	CHARDELETE
<end-of-line>	WAKEUPCHAR
↑Q	LINEDELETE
↑R	RETYPE
↑V	CTRLV

If any of these character is not available, the implementor should designate and document suitable replacements. All other characters should have terminal syntax class NONE.

deletion control message name	message
LINEDELETE	##<carriage-return>
1STCHDEL	\
NTHCHDEL	<the empty sequence>

---

12 INTERLISP-10 limits these character sequences to 5 characters.

POSTCHDEL \
EMPTYCHDEL ##<carriage-return>

The control field should be set to NIL. The deleted character echo mode field should be set to ECHO. The global echo mode field should be set to T. The lower-to-upper case conversion mode field should be set to NIL.

The control character echo modes should be set as follows:

character	control character echo mode
↑A	IGNORE
↑Q	IGNORE
↑R	IGNORE
↑V	UPARROW

The control character echo mode of <end-of-line> should be set so as to cause to the standard terminal in use to print subsequent characters on the line below the last, starting at the left-hand margin. If the sequence of characters used as the carriage return characters can be obtained singly, they must have echo mode REAL. The implementor may set the remaining control character echo mode fields at his own discretion (presumably being sensitive to the characters available and the hardware properties of the terminals used by prospective users).

```
TERMTABLEP[x]   If x is a Terminal Table, return x;
                else, return NIL.

GETTERMTABLE[termtbl]
                If termtbl is NIL, return the primary Terminal Table;
                elseif TERMTABLEP[termtbl], return termtbl;
                else, cause error 39 with culprit termtbl.

SETTERMTABLE[termtbl]
                If not TERMTABLEP[termtbl],
                  let termtbl be GETTERMTABLE[termtbl].
                Let oldtermtbl be the current primary Terminal Table.
                Set the primary Terminal Table to termtbl.
                Return oldtermtbl.

RESETTERMTABLE[termtbl;source]
                If not TERMTABLEP[termtbl],
                  let termtbl be GETTERMTABLE[termtbl].
                If not TERMTABLEP[source]:
                  If source = ORIG, let source be the original
                  Terminal Table;
                  else, let source be GETTERMTABLE[source].

                For every character, char, set the char terminal
                syntax class field in termtbl to the contents
                of that of char in source.

                For each deletion control message name, n, set
                the n deletion control message field in termtbl
                to the contents of that of n in source.

                Set the control field of termtbl to the contents of
                that of source.
```

```
                    For every control character, char, set the char control
                    character echo mode field in termtbl to
                    the contents of that of char in source.

                    Set the deleted character echo mode field of termtbl
                    to the contents of that of source.

                    Set the global echo mode field of termtbl to the
                    contents of that of source.

                    Set the lower-to-upper case conversion mode
                    field of termtbl to the contents of that of source.

                    Return termtbl.

COPYTERMTABLE[termtbl]
                    Let newtermtbl be a new Terminal Table.
                    Return RESETTERMTABLE[newtermtbl;termtbl].

ECHOCONTROL[char;mode;termtbl]
                    Let origchar be char.
                    If char is a Character or character code (treat
                    Characters 0 through 9 as character codes),
                      let char be the corresponding character;
                    else, cause error 27 with culprit char.

                    If char is not a control character:
                      If char is the equivalent of a control character,
                      let char be that control character;
                      else, cause error 27 with culprit origchar.

                    If not TERMTABLEP[termtbl],
                      let termtbl be GETTERMTABLE[termtbl].

                    Let oldmode be the contents of the char control
                    character echo mode field of termtbl.

                    If mode is NIL:
                      Return oldmode.
                    elseif mode is a control character echo mode:
                      Set the char control character echo mode field
                      of termtbl to mode.
                      Return oldmode;
                    else, cause error 27 with culprit mode.

DELETECONTROL[msgname;msg;termtbl]
                    If not TERMTABLEP[termtbl],
                      let termtbl be GETTERMTABLE[termtbl].
                    If msgname is DELETELINE,
                      let msgname be LINEDELETE.
                    If msgname is a deletion control message name:
                      If msg is NIL:
                        Create and return a String representing the
                        msgname deletion control message of termtbl;
                      elseif STRINGP[msg] or LITATOM[msg]:
                        If NCHARS[msg] is longer than the maximum
                        deletion control message length,
                          cause error 17 with culprit
                          CONS["Illegal message length - DELETECONTROL";msg].
                        Create a new string, oldmsg, representing
```

```
                    the msgname deletion control message of termtbl.
                    Set the msgname deletion control message
                    of termtbl to the character sequence in the
                    current pname of msg.
                    Return oldmsg;
                  else, cause error 17 with culprit
                  CONS["Illegal message type - DELETECONTROL";msg];
                elseif msgname is a deleted character echo mode:
                  Let oldmode be the current deleted character
                  echo mode of termtbl.
                  Set the deleted character echo mode field of termtbl
                  to msgname.
                  Return oldmode;
                else, cause error 27 with culprit msgname.

CONTROL[mode;termtbl]
                If not TERMTABLEP[termtbl],
                  let termtbl be GETTERMTABLE[termtbl].
                Let oldmode be the contents of the control field of termtbl.
                If mode, set the control field of termtbl to T;
                else, set the control field of termtbl to NIL.
                Return oldmode.

ECHOMODE[flg;termtbl]
                If not TERMTABLEP[termtbl],
                  let termtbl be GETTERMTABLE[termtbl].
                Let oldflg be the contents of the global echo mode
                field of termtbl.
                If flg is NIL, set the global echo mode field
                of termtbl to NIL;
                else, set the global echo mode field of termtbl to T.
                Return oldflg.

RAISE[flg;termtbl]
                If not TERMTABLEP[termtbl],
                  let termtbl be GETTERMTABLE[termtbl].
                Let oldflg be the contents of the lower-to-upper case
                conversion mode field of termtbl.
                If flg, set the lower-to-upper case conversion
                mode field of termtbl to T;
                elseif flg is NIL, set the lower-to-upper case conversion
                mode field of termtbl to NIL;
                else, set the lower-to-upper case conversion mode field
                of termtbl to 0.
                Return oldflg.
```

25. INTERRUPTS

As noted in the previous Section, it is desirable to provide the user with the ability to interrupt computational processes by typing special characters at the terminal. The VM provides a very flexible interrupt facility, based on *Terminal Assumptions* 1 and 2 (cf. Section 23).

Briefly, the user can associate "interrupt classes" with any of several "interrupt character codes". Whenever interrupts are "armed" (a condition under user control) and one of these

interrupt character codes is typed an appropriate "interrupt process" is invoked by the process defined in *Terminal Assumption* 2. Some of these processes provide handles for user specified computations. If an interrupt character is typed while interrupts are disarmed the character is ignored (as far as the system input buffer is concerned) and the interrupt process associated with that character is not invoked until interrupts are re-armed. The VM requires only that the last interrupt character typed while interrupts are disarmed be remembered for processing when interrupts are re-armed. Implementations may be more general, for example, by stacking interrupts while they are disarmed.

Recognizing that interrupts require special provisions in most operating systems and that often the available character codes are limited, the implementor is allowed to designate those character codes which may trigger interrupts.

Definition: A "valid interrupt character" is any character permitted by the implementation to trigger interrupts on terminal typein.

We will now describe the details of the interrupt capability.

The VM requires the existence of the following two fields to specify the state of the interrupt arm/disarm feature:

(1) the "interrupts armed" field, which contains either T or NIL, and

(2) the "saved interrupt character" field, which contains either NIL or a valid interrupt character.

When the interrupts armed field contains T we say interrupts are "armed". Otherwise they are "disarmed".

The initial contents of the interrupts armed field is T and the initial contents of the saved interrupt character field is NIL. These fields are used by the process defined in *Terminal Assumption* 2 and the function INTERRUPTABLE.

There is a third meta-object necessary to the specification of the interrupt facility and that is the "interrupt table". The interrupt table is somewhat like a Read or Terminal Table, in that it associates an "interrupt class" with each valid interrupt character.

Definition: A "basic interrupt class" is one of the Literal Atoms HELP, PRINTLEVEL, RUBOUT, ERROR, RESET, OUTPUTBUFFER, BREAK, ERRORX, INTERRUPT, or NONE.

Definition: The "interrupt table" is a meta-object such that for each valid interrupt character there is a field which contains either (1) a basic interrupt class, with the restriction that each of the first seven basic interrupt classes above may be in the field of at most one character, or (2) an arbitrary Literal Atom other than NIL or T.

Definition: The "interrupt class of (valid interrupt character) char" is the basic interrupt class or other Literal Atom in the char field of the interrupt table.

Each of the basic interrupt classes causes a certain VM specified process to be invoked when the associated characters are fetched from the terminal. If a character's interrupt class is a Literal Atom other than a basic interrupt class, the Literal Atom is used as a flag and set to T when the character is typed.

Before we can specify the processes invoked by these interrupts, we must clarify precisely when interrupts can occur.

It is not possible to interrupt an arbitrary process of the VM at an arbitrary point and then continue the interrupted process after an arbitrary VM computation. This is because the Virtual Machine is actually realized by an abstraction imposed upon a physical machine. The Virtual Machine is carried from one well-defined state to another by a series of "virtual steps" each of which is realized as a series of "actual steps" carried out by the physical machine. If the sequence of actual steps is interrupted at an arbitrary point the configuration of the physical machine may not correspond to the imposed abstraction, so that certain VM computations may not have any meaning (e.g., a function call when the stack is improperly configured due to the interruption).

We therefore assume that there are some implementor defined "safe" points at which arbitrary VM computations can be performed.

Definition: A "safe function call of fn on args" is a point during a computation at which the physical machine is in a "clean" state (one corresponding to a state of the Virtual Machine) and is about to do the equivalent of calling some function, fn, on some proper list of arguments, args.

The obvious safe calls are precisely those at which the Virtual Machine is about to execute a function call of a VM or user defined function fn on the proper list of arguments args. However, it is possible that there are additional safe states, depending on the implementation.

Interrupts are actually processed as soon as the corresponding interrupt character is typed (provided, of course, that interrupts are armed). Since interrupts involve computations entirely controlled by the implementor, it is assumed the interrupt handling can be done whether or not the physical machine is in a "clean" state. However, some interrupts provide the user with the illusion of being able to invoke an arbitrary user-specified computation at interrupt time. This, as we have seen, is not always meaningful. Therefore, at interrupt time these interrupts merely store sufficient information to cause the user-specified computation to be performed at the next safe point.

Some of the specifications for the interrupt processes below involve VM function calls, such as "CLEARBUFF[T]". It may not always be possible to execute such calls in the manner non-VM function calls would be executed (i.e., by building frames on the user stack), given the arbitrary state of the physical machine at the time the interrupt character is fetched from the terminal. What is meant by these specifications is that the actions specified for the called VM function should be performed (and the precise mechanism of the call is unspecified). If a non-VM function is to be called by an interrupt process, the specifications will explicitly say that the stack should be destructively backed up to an acceptable state (i.e., any partially constructed frame at the end of the stack is removed and the user is to understand that this represents an irrecoverable detour in the flow of control).

One of the interrupt classes, PRINTLEVEL, uses special auxilary buffers in which to save the contents of the line and system buffers while interacting with the user. This requires the existence of two distinct buffers.

Definition: The "interrupt line buffer" is a buffer of the same length as the line buffer[13]. The "interrupt system buffer" is a buffer of the same length as the system buffer. These four buffers are all distinct.

The following definition allows us to "copy" one buffer from another (and empty the first in the process).

Definition: To "copy buff1 to buff2" (where buff1 and buff2 are buffers of the same length) means:

```
"Clear buff2.
 Until buff1 is empty, fetch characters from buff1
 and deposit them in buff2."
```

We can now specify the processes associated with each interrupt class. Recall *Terminal Assumption* 2. Let char be the character just fetched from the terminal. Assume char is a valid interrupt character, not preceded by the CTRLV character, assume interrupts are armed, and let charcode be the character code of char. Finally, let classname be the interrupt class for char. Then the process invoked when char is typed is specified below, according to classname:

```
classname                Process

  HELP                   Clear the system output buffer.
                         Send the bell character to the terminal.
                         CLEARBUF[T].
                         Save sufficient information so that
                         INTERRUPT[fn;args;1] is evaluated at
                         the next safe function call of some
                         function fn on argument list args.

  PRINTLEVEL             Copy the line buffer to the interrupt
                         line buffer.
                         Copy the system buffer to the interrupt system
                         buffer.
                         Send the bell character to the terminal.

                         Let seq be the character sequence obtained
                         by fetching characters directly from the
                         terminal up to and including the first
                         character which is neither a <digit> (cf. Section 9)
                         nor a ',' (comma).

                         Let lastchar be the last character fetched.
                         Let seq be a new sequence obtained by removing
                         the last character in seq.
                         If (the new) seq contains a ',':
                           Let carval be the integer denoted by
                           the digit sequence to the left of the
                           ',' in seq (with the empty sequence
                           denoting 0).
                           Let cdrval be the integer denoted by
                           the digit sequence to the right of the
```

---

13 The "line buffer" is defined in Section 27.

'.' in seq (with the empty sequence
denoting 0).
else:
Let carval be the integer denoted by
seq (with the empty sequence
denoting 0).
Let cdrval be NIL.

If lastchar is '!':
Set the temporary car print level field
to carval (cf. Section 26).
Set the car print level field to
carval.
Set the temporary cdr print level field
to cdrval.
Set the cdr print level field to
cdrval.
elseif lastchar is '.':
Set the temporary car print level field
to carval.
If cdrval, set the temporary cdr
print level field to cdrval.

Copy the interrupt system buffer to the
system buffer.
Copy the interrupt line buffer to the
line buffer.
Continue the interrupted computation
without further change of state.

RUBOUT

Clear the system input buffer.
Send the bell character to the terminal.

Continue the interrupted computation
without further change of state.

ERROR

Clear the system output buffer.
Send a carriage return character to
the terminal.
CLEARBUF[T].
If the interrupted process was adding
a new frame to the stack, clear off
any uncompleted frame (thereby backing
the stack up to the last completed function
call and allowing a normal non-VM function
call to be executed).
ERROR![].
Note: If the interrupted process is any VM
process which, if terminated prematurely, is
liable to leave the VM in a meaningless state
(such as a garbage collector or storage compactor
might) the execution of ERROR![] should be delayed
until the process has terminated normally.

RESET

Clear the system output buffer.
Send a carriage return character to
the terminal.
CLEARBUF[T].
If the interrupted process was adding
a new frame to the stack, clear off
any uncompleted frame (thereby backing

	the stack up to the last completed function call and allowing a normal non-VM function call to be executed). RESET[]. Note: If the interrupted process is any VM process which, if terminated prematurely, is liable to leave the VM in a meaningless state (such as a garbage collector or storage compactor might) the execution of RESET[] should be delayed until the process has terminated normally.
OUTPUTBUFFER	Clear the system output buffer. Send the carriage return character to the terminal. Continue the interrupted computation without further change of state.
BREAK	Clear the system output buffer. CLEARBUF[T]. If the interrupted process was adding a new frame to the stack, clear off any uncompleted frame (thereby backing the stack up to the last completed function call and allowing a normal non-VM function call to be executed). Cause error 18 with culprit NIL. Note: If the interrupted process is any VM process which, if terminated prematurely, is liable to leave the VM in a meaningless state (such as a garbage collector or storage compactor might) delay causing the error until the process has terminated normally.
ERRORX	Clear the system output buffer. CLEARBUF[T]. If the interrupted process was adding a new frame to the stack, clear off any uncompleted frame (thereby backing the stack up to the last completed function call and allowing a normal non-VM function call to be executed). Cause error 43 with culprit charcode. Note: If the interrupted process is any VM process which, if terminated prematurely, is liable to leave the VM in a meaningless state (such as a garbage collector or storage compactor might) delay causing the error until the process has terminated normally.
INTERRUPT	Clear the system output buffer. Send the bell character to the terminal. CLEARBUF[T]. Save sufficient information so that INTERRUPT[fn;args;charcode+65] is evaluated at the next safe function call of some function fn on argument list args.
NONE	Cause no interrupt.
all other classes	Immediately perform SETQ[classname;T], where classname is the Literal Atom which

```
                    is the name of the interrupt class of charcode.
                    Continue the interrupted computation without
                    further change of state.
```

The functions for manipulating the state of the interrupt armed field and the interrupt table are specified below.

```
INTERRUPTABLE[flg1:...flgk]
                Let oldflg be the contents of the interrupts
                armed field.

                If k > 0:
                  If flg1,
                    Set the interrupts armed field to T.
                    If the saved interrupt character field contains a
                    character, char, (rather than NIL) with interrupt
                    class, class:
                      Set the saved interrupt character field to NIL.
                      Perform the computation specified for the class
                      interrupt.
                  else, set the interrupts armed field to NIL.

                Return oldflg.
```

Note: The purpose of the saved interrupt character field is to insure that if an interrupt character is typed while interrupts are disarmed then the last such interrupt is processed once interrupts are re-armed. The implementor may choose a more general regime, such as stacking or queuing the interrupts (even though that means that the last interrupt may not be processed because an earlier one aborted the entire computation).

```
GETINTERRUPT[char]
                Let origchar be char.
                If char is a Character or character code
                (treat Character 0 through 9 as character codes),
                  let char be the corresponding character.

                If char is an interrupt class,
                  create and return a new proper list of all of the
                  valid interrupt character codes which have char in
                  their field of the interrupt table;
                elseif char is a valid interrupt character,
                  return the contents of the char field of the
                  interrupt table;
                else, cause error 27 with culprit origchar.

SETINTERRUPT[char;class]
                Let origchar be char.
                If char is a Character or character code
                (treat Character 0 through 9 as character codes),
                  let char be the corresponding character.

                If char is not a valid interrupt character,
                  cause error 27 with culprit origchar.
                If class = NIL or class = T,
                  cause error 27 with culprit class.
                Let oldclass be the contents of the char field of the
                interrupt table.
```

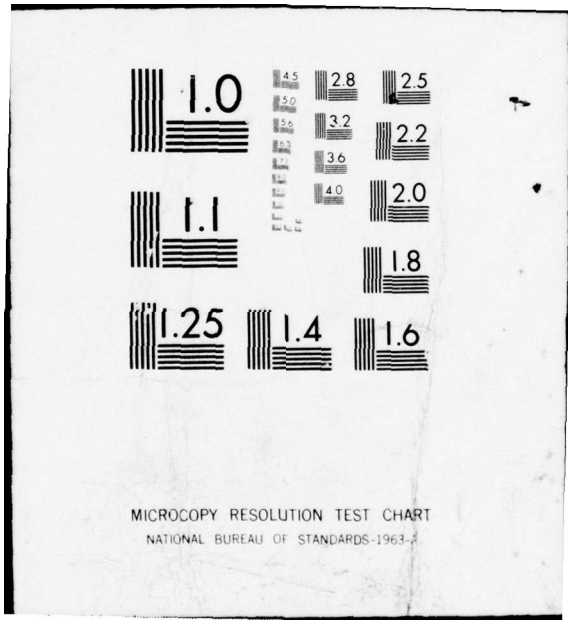
AD-A036 468 XEROX PALO ALTO RESEARCH CENTER CALIF COMPUTER SCIEN--ETC F/G 9/2
THE INTERLISP VIRTUAL MACHINE SPECIFICATION,(U)
SEP 76 J S MOORE N00014-75-C-0626

UNCLASSIFIED NL

2 OF 2
AD
A036468

END

DATE
FILMED
3-77


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```
If class is a basic interrupt class other than
ERRORX, INTERRUPT, or NONE:
  If any valid interrupt character, c, has
  class in its field of the interrupt table,
    set the c field of the interrupt table to NONE.
  Set the char field of the interrupt table to class.
  Return oldclass;
else:
  Set the char field of the interrupt table to class.
  Return oldclass.
```

26. OUTPUT

The output routines are responsible for transferring characters from the VM to the terminal and other files. These routines therefore translate objects into character sequences.

Because almost every function in this Section deals with a file, a Read Table, and the primary Terminal Table, the following definitions are useful.

Definition: To "check File Name file for output" where file denotes a meta-variable which denotes an object, means:

```
"Let obj be the object denoted by file.
If obj is NIL, let file be the primary output file;
elseif obj is T, let file be T (no-op);
elseif OPENP[obj;OUTPUT], let file be OPENP[obj;OUTPUT];
else, cause error 13 with culprit obj."
```

We also assert the analogous definition for "check File Name file for input" (replace "output" above by "input", and replace OUTPUT by INPUT).

Note that after "check File Name x for output" is used, x denotes the full File Name of a file open for output (or else an error was caused).

Definition: To "check Read Table rdtbl" where rdtbl is a meta-variable denoting a meta-variable which denotes an object, means "Let obj be the object denoted by rdtbl. If not READTABLEP[obj], let rdtbl be GETREADTABLE[obj]."

Convention: All references to a Terminal Table refer to the primary Terminal Table at the time of the operation specified. Example: "the terminal syntax class of char" means "the contents of the char terminal syntax class field in the primary Terminal Table."

Convention: Any operation which requires a file or Read Table but does not specify one will implicitly refer to some file or Read Table which was distinguished earlier in the same definition or specification by the phrase "use File Name (or Read Table) x implicitly below."

The most basic definable output operation is that of writing a single character to a specified file. This relies upon the primitive idea of "depositing" a character in a specified file. However writing is complicated by the possible involvement of a Terminal Table.

Definition: To "write (character) char to (file) file" means:

```
"If file is T:
  If char is a control character:
    Let mode be the control character echo mode of char.
    If mode = REAL, deposit char in the terminal;
    elseif mode = SIMULATE, invoke the control character simulation
    procedure for char (see Note below);
    elseif mode = UPARROW:
      Deposit the character '↑' in the terminal.
      Deposit the ↑equivalent of char in the terminal.
  else, deposit char in the terminal.
  If there is a dribble file,
    write character char to the dribble file;
elseif file is an addressable file:
  Let i be the file pointer of file.
  Deposit char in the ith character field of file.
  Set the file pointer of file to i+1;
else, deposit char in file file."
```

Note: We assume that for each control character there is a simulation procedure which computes some sequence of characters and writes them successively to the file in question. The precise sequence computed for any control character in any situation is unspecified. It is supposed that the sequence attempts to immitate, when possible, the formatting function normally performed by the control character.

The following trivial extension of our terminology is useful.

Definition: To "write (character sequence) seq to file" means to "write the successive characters in seq to file."

It is convenient to allow the user to globally specify certain parameters influencing output. These include the maximum allowable line length, the depth to which List Structures are printed, the length to which List Structures are printed, and the radix used to print integers. These parameters are held in fields accessible to the user through certain functions.

The VM requires the existence of the following six fields:

(1) There is a field, called the "line length" field, which contains an integer.

(2) There is a field, called the "car print level" field, which contains an integer.

(3) There is a field, called the "temporary car print level" field, which contains an integer.

(4) There is a field, called the "cdr print level" field, which contains an integer.

(5) There is a field, called the "temporary cdr print level" field, which contains an integer.

(6) There is a field, called the "radix" field, which contains an integer in an implementor specified range which must include 2 through 10 and may include other integers (see discussion below).

The output functions specified below (PRIN1, PRIN2, PRIN3, PRIN4, SPACES, TERPRI and PRINT) use these fields to control printing. The functions LINELENGTH, PRINTLEVEL and RADIX are available for accessing/replacing these fields.

The initial contents of the line length field is to be determined by the implementor, with due regard to the line length of the standard terminal in use. The car print level field initially contains 1000. The cdr print level field initially contains -1. The contents of the two temporary print level fields are unspecified, since they are always initialized before use. The radix field initially contains 10. The contents of the radix field determines the base in which Integers are printed. The following definition specifies the restrictions on the contents of the radix field by defining the relationship between the contents of the radix field and the character sequence used to output an Integer.

Definition: By "let seq be the base-r representation of (Integer) i", where seq denotes a meta-variable, r is an integer which the implementor allows to be in the radix field and i is an Integer, we mean the following:

```
"If 2=<r=<10 (the implementor must allow this range in the radix):
  Let seq be the standard base-r notation for the integer i,
  employing the usual digits 0 through r-1, being explicitly
  signed (with '-') only if negative, and having leading 0's removed;
elseif r > 10 (for this to occur the implementor
must have chosen a set of characters
to be used as digits beyond '9' allowing consistent
notation in base-r):
  Let seq be the standard base-r notation for the integer i,
  employing the implementors extended alphabet and
  following the sign and 0 conventions above;
else (r is negative, the implementor allows representation
in base-|r|, and the host machine uses complements representation
of integers):
  If i<0:
    Let b be the bit pattern representing the integer i in the host machine.
    Let i' be the positive integer with binary expansion b
    (i.e., with no special interpretation of the sign bit).
    Let seq be the base-|r| representation of i';
  else, let seq be the base-|r| representation of i;"
```

We assume that the notion of balanced (or "matched") parentheses is understood.

The following function, PRIN1, is the basic output function in the VM. The specification of PRIN1 is recursive: PRIN1 is called several times from within the specification. We assume that the notion of the "top-level" call is understood to mean an invocation not contained within the specification below.

```
PRIN1[x;file]   Check File Name file for output and use file implicitly below.
                Let pos be the contents of the position field of file.
                Let lnlen be the contents of the line length field.
                Set the temporary car print level field to the contents
                of the car print level field.
                Set the temporary cdr print level field to the contents
                of the cdr print level field.
                If file is T, set plvlflg to T;
                elseif PLVLFILEFLG, set plvlflg to T;
                else, set plvlflg to NIL.
                (Note:  plvlflg is set to T when PRIN1 is to be
                sensitive to the print level fields -- i.e., terminate
                the printing of lists after a certain depth/length.
                The value of the Literal Atom PLVLFILEFLG determines
                whether the print levels are to influence output to
                files other than the terminal.)
```

```
If LITATOM[x]:
  Let n be the number of characters in the name of x.
  If lnlen>=0 and pos+n>lnlen,
    write a carriage return character.
  Write the name of x.
  Return x;
elseif FIXP[x]:
  Let r be the contents of the radix field.
  Let seq be the base-r representation of x.
  Let n be the number of characters in seq.
  If lnlen>=0 and pos+n>lnlen,
    write a carriage return character.
  Write seq.
  Return x;
elseif FLOATP[x]:
  Let seq be a character sequence defined by
  <floating point number> (cf. Section 10)
  denoting the real represented by x.
  (The implementor is allowed to choose the
  form of seq desired.)
  Let n be the number of characters in seq.
  If lnlen>=0 and pos+n>lnlen,
    write a carriage return character.
  Write seq
  Return x;
elseif STRINGP[x]:
  Let seq be the character sequence represented by x.
  Write seq.
  Return x;
elseif LISTP[x]:
  If plvlflg and the number of unmatched '('s
  printed by the "Write '('." statement (see below)
  thus far under the top-level call to PRIN1 is equal
  to or greater than the absolute value of the
  contents of the temporary car print level field:
    Write '&'.
    Return x;
  elseif plvlflg and the contents of the temporary
  car print level field is negative and the last character
  printed by PRIN1 was the ')' written by the "Write ')'."
  statement below,
    write the carriage return character.

  Let origx be x.
  Let cdrcnt be 0.
  Write '('.
  (The above "Write" increments the current number
  of unmatched '('s printed thus far.)
  Until "x has been printed" (defined below), do the
  following:
    (x is reset during this "Until" loop.)
    If plvlflg and the contents of the temporary cdr
    print level field is equal to cdrcnt:
      Write '--'.
      We now say that x has been printed and the
      "Until" loop should be immediately exited.
    elseif not LISTP[x]:
      Write '.'.
      Write ' '.
      Decrement the contents of the temporary cdr print
      level field by 1 and store the results back in the
      temporary cdr print level field.
```

```
        PRIN1[x;file].
        Increment the contents of the temporary cdr print
        level field by 1 and store the results back in the
        temporary cdr print level field.
        We now say that x has been printed and the
        "Until" loop should be immediately exited.
      else.
        Decrement the contents of the temporary cdr print
        level field by 1 and store the result back in the
        temporary cdr print level field.
        PRIN1[CAR[x];file].
        Increment the contents of the temporary cdr print
        level field by 1 and store the results back in the
        temporary cdr print level field.

      Let x be CDR[x].
      If x is NIL, we say x has been printed and the
      "Until" loop should be immediately exited.

      Write ' '.
      If plvlflg and the number of unmatched '('s
      printed by the "Write '('." statement (see above)
      thus far under the top-level call to PRIN1 exceeds
      the absolute value of the contents of the temporary
      car print level field:
        Write "--".
        We say that x has been printed and the "Until"
        loop should be immediately exited;
      else, continue the "Until" loop.
    Write ')'.
    (This "Write" decrements the current number of
    unmatched '('s printed thus far.)
    Return origx;
  else, write some (unspecified) sequence of characters.
```

Note that for objects other than Literal Atoms, Numbers, Strings, and List Cells the VM does not specify the sequence of characters printed (beyond the requirement that some sequence be printed).

```
PRIN2[x;file;rdtbl]
        Check File Name file for output and use file implicitly below.
        Check Read Table rdtbl and use rdtbl implicitly below.
        Let pos be the contents of the position field of file.
        Let lnlen be the contents of the line length field.
        Set the temporary car print level field to the contents of
        the car print level field.
        Set the temporary cdr print level field to the contents of
        the cdr print level field.
        If file is T, set plvlflg to T;
        elseif PLVLFILEFLG, set plvlflg to T;
        else, set plvlflg to NIL.
        (Note:  plvlflg is set to T when PRIN1 is to be
        sensitive to the print level fields -- i.e., terminate
        the printing of lists after a certain depth/length.
        The value of the Literal Atom PLVLFILEFLG determines
        whether the print levels are to influence output to
        files other than the terminal.)

        If LITATOM[x]:
          Let seq be the character sequence formed from
          the name of x by placing a '<percent>' character
```

```
                    immediately before every character, c,
                    in the name of x such that:
                      c is a break character, a separator character,
                      or an ESCAPE character, or c is a read macro whose
                      escape flag component is ESCQUOTE.
                    Let n be the number of characters in seq.
                    If lnlen>=0 and pos+n>lnlen,
                      write a carriage return character.
                    Write seq.
                    Return x;
                  If FIXP[x]:
                    PRIN1[x;file].
                    If the contents of the radix field is 8, write 'Q'.
                    Return x;
                  elseif FLOATP[x], PRIN1[x;file];
                  elseif STRINGP[x]:
                    Let seq be the character sequence formed from
                    the character sequence represented by
                    x by placing a '<percent>' character
                    immediately before every character, c,
                    in the character sequence represented by
                     x such that:
                      c is a break character, a separator character,
                      or an ESCAPE character, or c is a read macro whose
                      escape flag component is ESCQUOTE.
                    Write '"'.
                    Write seq.
                    Write '"'.
                    Return x;
                  elseif LISTP[x]:
                    (Same specification as in the LISTP[x] clause
                    in PRIN1 except that "PRIN2" should be used instead
                    of "PRIN1" (and "rdtbl" should be added as a third
                    argument in those PRIN2 calls).
                  else, write some (unspecified) sequence of characters.


PRIN3[x;file;]  (Same specification as for PRIN1 except that
                all statements involving the meta-variable pos
                and all statements involving the meta-variable lnlen
                are left out.)

PRIN4[x;file;rdtbl]
                (Same specification as for PRIN2 except that
                all statements involving the meta-variable pos
                and all statements involving the meta-variable lnlen
                are left out.)

SPACES[n;file]  If n=NIL, let n be 1;
                elseif not FIXP[n], let n be FIX[n].

                If n<0, let n be 0.

                Let str be a String of length n, containing n
                space (i.e., ' ') characters.
                PRIN1[str;file].
                Return NIL.
```

Note: SPACES is used so frequently it is best implemented so as to avoid actually constructing a new String str.

```
TERPRI[file]    Let str be a String consisting of a single
                carriage return character.
                PRIN1[str;file].
                Return NIL.

PRINT[x;file;rdtbl]
                PRIN2[x;file;rdtbl].
                TERPRI[file].

LINELENGTH[n]   Let oldn be the representation as an
                Integer of the contents of the line length field.
                If n is NIL, return oldn.
                If not FIXP[n], let n be FIX[n].
                Set the line length field to the integer represented by n.
                Return oldn.

PRINTLEVEL[carval;cdrval]
                Let oldval be CONS[oldcarval;oldcdrval], where oldcarval
                is the representation as an Integer of the contents of
                the car print level field, and oldcdrval is the
                representation as an Integer of the contents of the cdr
                print level field.

                If LISTP[carval]:
                  Let cdrval be CDR[carval].
                  Let carval be CAR[carval].

                If carval is not NIL:
                  Set the car print level field to the integer represented
                  by FIX[carval].
                If cdrval is not NIL:
                  Set the cdr print level field to the integer represented
                  by FIX[cdrval].

                Return oldval.

RADIX[n]        Let oldn be the representation as an
                Integer of the contents of the radix field.
                If n is NIL, return oldn.
                If not FIXP[n], let n be FIX[n].
                If the implementation does not allow
                the radix field to contain n (i.e., the implementation
                does not allow base-n representation as defined above),
                  cause error 27 with culprit n.
                Set the radix field to the integer represented by n.
                Return oldn.
```

27. INPUT

The input routines are responsible for transferring characters from files into the VM. These routines parse the incoming sequences into objects, according to information contained in Read and Terminal Tables available at the time of the transfer.

The most basic definable input operation is reading the "next" character from a specified file. We rely on the analogous primitive operation, that of "fetching" the next character from a

dynamic stream from some device, or of "fetching" the character at some address of a stored file. Reading and fetching are distinguished because the former must give the latter a file pointer from which to fetch, and must interpret the resulting character in light of the Read Table in use.

The process of reading from the terminal is even more complicated due to the involvement of a Terminal Table. We introduce the idea of the "line buffer" to define the operation of reading from the terminal. Then we will return to the problem of reading from files in general.

Definition: The "line buffer" is a buffer of unspecified length, used to hold characters obtained from the system input buffer while they are still subject to user controlled editing operations.

Recall that we assume that characters from the terminal are continuously being deposited into the system input buffer. Whenever an input request is made by the VM, characters are fetched from the system input buffer and deposited in the line buffer as described below. Two buffers are required for two reasons: The system buffer must continue to accept characters even while the editing operations or other computational processes occupy the line buffer (thus, the system buffer is at a very low level), and the method by which the line buffer is filled from the system input buffer is very sensitive to computational context within the VM (thus, the line buffer is at a very high level).

Convention: Because the system input buffer is usually below the level we need for our specifications, we will henceforth refer to the line buffer simply as "the buffer".

Definition: To "fill the buffer until p", where p is some statement describing a situation means:

```
"In the following, use the system input buffer implicity
for all fetches, the terminal for all writes, and whatever
Read Table has been distinguished as the one in use.

Until the buffer is "full" (defined below), do the following:
  If the buffer is replete,
    the buffer is said to be "full" and the filling process
    is complete, and the "Until" should be immediately exited.

  Fetch the next character, char.

  If char is lower case and the lower-to-upper case conversion
  mode field is 0, let char be the corresponding upper case character.
  If the global echo mode is T, write char.
  Let class be the terminal syntax class of char.

  If statement p is true or class = WAKEUPCHAR or char is a read
  macro (in the distinguished Read Table) with wakeup mode WAKEUP:
    If char is lower case and the lower-to-upper case conversion
    mode is T, let char be the corresponding upper case character.
    Deposit char in the buffer.
    The buffer is now said to be "full", the filling process
    is complete, and the "Until" should be immediately exited;
  elseif class = CHARDELETE
    If the buffer is empty:
      Write the EMPTYCHDEL deletion control message;
    else:
      Let oldchar be the character code fetched from T immediately
      before char was fetched (note that this is
      not necessarily the last character in the buffer).
      If oldchar was the CHARDELETE character code,
```

```
        write the NTHCHDEL deletion control message;
      else, write the 1STCHDEL deletion control message.
      If the deleted character echo mode is T:
        Let char' be the character in the line buffer field
        indicated by the deposit pointer.
        Write char'.
        If char' was preceded when fetched by an ESCAPE character, escchar,
          write escchar.
      Decrement the line buffer deposit pointer field by 1
      and store the result in the line buffer deposit pointer field.
      If the next character to be fetched is not the
      CHARDELETE character, write the POSTCHDEL deletion control message;
  elseif class = LINEDELETE:
    Write the LINEDELETE deletion control message.
    Clear the buffer;
  elseif class = RETYPE:
    Write the carriage return character.
    Write the character sequence currently corresponding to the buffer;
  elseif class = CTRLV:
    Fetch the next character, char.
    If char is the ↑equivalent of some control character,
      let char be the control character;
    elseif char is lower case and the lower-to-upper case
    conversion mode is T,
      let char be the corresponding upper case character.
    Deposit char in the buffer;
  else (class must be NONE):
    If char is lower case and the lower-to-upper case conversion
    mode is T, let char be the corresponding upper case character.
    Deposit char in the buffer.
  Continue the "Until"."
```

Convention: If no statement p is supplied for a filling operation, p is assumed to be always false. If we say "filling until T", we mean p is considered always true, which is equivalent to saying deposit exactly one character in the buffer and then consider it "full".

Of course, in general, the statement p just allows higher level routines to specify conditions under which the buffer should be considered to be full (e.g., when a matching right parenthesis is fetched).

Note that whether the buffer is "full" depends upon the process which controlled filling it. The buffer is "replete" when every field in it contains a character. This condition is independent of the controlling process.

Once the buffer is full, we think of it as a queue of characters, precisely like the system input buffer.

We can now specify what it means to "read" a character from a specified file.

Definition: To "read a (or the next) character, char, from (File Name) file (filling until p)", where char denotes a meta-variable means:

```
"If file is T:
  If the line buffer is empty:
    If the control mode is T, fill the buffer until p;
    else, fill the buffer.
  Fetch the next character, c, from the buffer.
  If there is a dribble file, write character
```

```
    c to the dribble file.
  elseif file is an addressable file:
    Let i be file pointer of file file.
    If i is equal to or greater than the end of file pointer of file:
      CLOSEF[file].
      Cause error 16 with culprit file.
    Fetch the character, c, in field i of file.
    Set the file pointer of file to i+1.
  else, fetch the next character, c, from file.
  Let char be c."
```

Note that if the file is T, the character is actually fetched from the line buffer. Note also that the parenthetical clause specifying the statement p has no effect if the file is other than T, or if the control mode in the primary Terminal Table is NIL.

We have now formally specified the effect of every field in a Terminal Table. We have also defined the two basic input/output operations, upon which the VM LISP functions can be based.

Definition: The following rather uncomfortable phrase: "Let seq be the sequence of characters obtained by reading from file (filling until p) until q", where seq denotes a meta-variable, means "Let seq be the sequence of successive characters obtained by reading characters from file repeatedly until condition q is satisfied (with all fill operations to be done being governed by statement p)."

The following definition specifies when a read macro character's syntactic context permits invocation of the body of the read macro.

Definition: If char is the first character of a character sequence we say that "char is a *read macro in its context" if the following three statements are true:* char has a read macro 5-tuple, <type, context, wakeup mode, escape flag, body>, in its syntax class field, the read macros enabled field of the Read Table in use contains T, and either (1) context is ALWAYS or FIRST, or (2) context is ALONE, and the following character of the sequence is a break or separator character.

Note (by inspection of the definition of "fill the buffer until p") that if a read macro character has wakeup mode WAKEUP the line buffer is considered full (and may be processed by the read routines defined below) as soon as the character has been deposited into the buffer. Usually, this would mean the macro would be expanded as soon as the READ routine sees it in the line buffer. But if its context is ALONE it is not possible to determine whether the read macro is in its context until the next fill operation has been completed (making the following character available). Thus, its expansion would be delayed.

It is convenient if programs can discover whether they are executing "under" a call to a read macro. Therefore, whenever a read macro is evaluated, the frame extension associated with that activation is marked (in the temporaries field). There are two kinds of marks: one denoting an "armed" call to a read macro, and one denoting an "unarmed" call to a read macro. The difference is that certain situations cause errors when they occur under armed calls, and do not cause errors under unarmed calls. The user can change the mark associated with a read macro activation by using the function SETREADMACROFLG (defined below). Both the functions READ and INREADMACROP inspect these marks.

Convention: We assume the notion of balanced parentheses is well-defined. We extend it to

the characters in the syntax classes LEFTPAREN and RIGHTPAREN by merely considering (for the purposes of matching or balancing characters) every character in the first to be a left parenthesis and every character in the second to be a right parenthesis.

We will define the functions of the ESCAPE, LEFTBRACKET, and RIGHTBRACKET syntax classes below. Essentially, ESCAPE allows the next character to be treated as though it had syntax class OTHER. The LEFTBRACKET and RIGHTBRACKET classes contain "super-parentheses".

Definition: By "observe the ESCAPE guidelines" we mean:

```
"In the following, if an ESCAPE character is ever
fetched into the line buffer, the character should be ignored, the next
character, char, should be fetched and used in its place, and
that occurrence of char should be treated as though it had syntax
class OTHER."
```

Definition: By "observe the LEFTBRACKET and RIGHTBRACKET guidelines" we mean:

```
"In the following, if a LEFTBRACKET character, char,
is fetched into the line buffer:
  We say that it is an "unmatched LEFTBRACKET" (until a matching
  RIGHTBRACKET character is fetched).
  Treat this occurrence of char as though it were a LEFTPAREN.
In the following, if a RIGHTBRACKET character, char, is fetched:
  If a still unmatched LEFTBRACKET has been fetched:
    (Below we consider char to denote the occurrence of the RIGHTBRACKET.)
    Let char' be the last unmatched occurrence of a LEFTBRACKET.
    (We say that char' is now matched.)
    Let n be the number of unmatched LEFTPAREN characters fetched
    between char' and char.
    Treat char as though it were a RIGHTPAREN character, followed by
    n additional RIGHTPAREN characters;
  else:
    Let n be the number of still unmatched LEFTPAREN characters fetched.
    If n is 0, treat char as though it were a RIGHTPAREN character;
    else, treat char as though it were a RIGHTPAREN character
    followed by n-1 RIGHTPAREN characters."
```

The function below, READ, is recursive. As for PRIN1 we assume the notion of the top-level call is understood to be an invocation of READ not contained within the specification of READ. Note that the state of the file being read at the time of the top-level call to the function determines certain behavior exhibited by all of the recursive calls.

```
READ[file;rdtbl;flg]
                Check File Name file for input and use file implicitly below;
                Check Read Table rdtbl and use rdtbl implicitly below.

                In the following, any read macro character with
                context ALWAYS is to be treated as though it
                were a break character.

                In the following, every fill operation is to
                keep track of the balanced LEFTPAREN, LEFTBRACKET,
                RIGHTPAREN, and RIGHTBRACKET characters from
                the top-level call of READ, and all filling operations
```

```
are to be done as though the control field of the
primary Terminal Table were T and until a matching or
unmatched RIGHTPAREN or RIGHTBRACKET is fetched into the
line buffer.  At the time that character is fetched,
if file is T and flg is non-NIL, the carriage return
character is to be written to T.

Observe the ESCAPE, LEFTBRACKET, and RIGHTBRACKET guidelines.

Let char be the next character to be read.
If char is a separator character, read characters until
the next character, char, is not a separator character.

If char is a STRINGDELIM:
  Read character char and ignore it.
  Let seq be the sequence of characters obtained by reading
  until the next character to be read is a STRINGDELIM
  (and treat LEFTPAREN, LEFTBRACKET, RIGHTPAREN,
  and RIGHTBRACKET characters as though they had
  syntax class OTHER -- i.e., do not consider them for
  the purposes of balancing).
  Read the final STRINGDELIM and ignore it.
  Create and return a new String with pname seq.
elseif char is an (unmatched) RIGHTPAREN:
  Return NIL;
elseif char is a LEFTPAREN:
  Read the LEFTPAREN and ignore it.
  Assemble a new List Structure in the following way:
    (We say a List Structure is being assembled.)
    Let anscell be CONS[NIL;NIL].
    (anscell will be used as what the INTERLISP
    Reference Manual calls a TCONC-list:  its CAR
    will generally contain a proper list
    and its CDR will contain the last list cell in the
    CDR-chain of that proper list.)

    Until the next character, char, to be read
    (by any recursive call to READ below)
    is the RIGHTPAREN matching the LEFTPAREN
    just read, do the following:
      If char is a SPLICE type read macro
      in its context:
        Read char.
        Let macval be the result of evaluating
        body[file;rdtbl], where body is the
        value of the body attribute of read macro char,
        in a frame extension marked as armed
        (see Note below).
        If macval is a List Cell:
          Let lastcell be the last List Cell in the CDR
          chain of macval.
          Let lastcdr be CDR[lastcell].
          If lastcdr is non-NIL:
            (Then macval is a not a proper list.)
            Make macval a proper list by replacing
            the CDR of lastcell with a new proper list
            of length 2 with the Literal Atom . as
            its first element and lastcdr as its second;
        elseif macval is non-NIL:
          (Then macval is not a proper list.)
          Make macval a proper list by letting
          macval be a new proper list of length 2 with
```

```
            the Literal Atom . as its first element
            and macval as its second.
          (Now macval denotes a proper list.)
          If macval is non-NIL:
            Let lastcell be the last List Cell in the
            CDR chain of macval.
            If CDR[anscell] is NIL:
              RPLACA[anscell;lastcell].
              RPLACD[anscell;lastcell];
            else:
              RPLACD[CDR[anscell];lastcell].
              RPLACD[anscell;lastcell];
        elseif char is an INFIX type read macro in its
        context:
          Read char.
          Let macval be the result of evaluating
          body[file;rdtbl;anscell], where body
          is the value of the body attribute of read
          macro char, in a frame extension marked as armed
          (see Note below).
          Assume macval is a List Cell whose CAR
          contains a proper list and whose CDR contains the last
          List Cell in the CDR chain of that proper list.
          Let anscell be macval.
        else:
          (We will not make a special case
          above for reading MACRO type read macros
          while assembling List Structures.
          These are handled outside the context of
          assembling List Structures (below) and are
          handled inside List Structures without further
          special consideration by the READ call in
          this "else" clause.)
          Let macval be CONS[READ[file;rdtbl];NIL]
          (note recursion).
          If CDR[anscell] is NIL:
            RPLACA[anscell;macval]
            RPLACD[anscell;macval];
          else:
            RPLACD[CDR[anscell];macval].
            RPLACD[anscell;macval].
      Read the matching RIGHTPAREN and ignore it.
      Let ans be CAR[anscell].
      (ans should be a proper list.)
      If ans has more than 2 elements:
        Let lastcells be the second from the last List Cell
        in the CDR chain of ans.
        If CAR[lastcells] is the Literal Atom . and was
        either (1) produced by a recursive call to READ
        (in the else-clause of the Until above)
        which read the character '.' not preceded
        by an ESCAPE or (2) was one of the two occurrences
        of . introduced in order to make macval a proper list
        (in the SPLICE read macro clause above),
        perform RPLACD[lastcells;CAR[CDR[lastcells]]].
      (We have now completed assembling the List Structure.)
      Return ans;
    elseif char is a read macro in its context:
      Read char.
      If the type of char is MACRO:
        Let macval be the result of evaluating
        body[file;rdtbl], where body is the
```

```
        value of the body attribute of read macro char,
        in a frame extension marked as armed (see Note below).
        Return macval;
      elseif the type of char is SPLICE:
        (Note that occurrences of SPLICE read macros within
        List Structures is handled above.)
        Compute and ignore the result of evaluating
        body[file;rdtbl], where body is the
        value of the body attribute of read macro char,
        in a frame extension marked as armed (see Note below).
        Return READ[file;rdtbl];
      else (the type of char is INFIX):
        (Note that occurrences of INFIX read macros within
        List Structures is handled above.)
        Let macval be the result of evaluating
        body[file;rdtbl;NIL], where body is the
        value of the body attribute of read macro char,
        in a frame extension marked as armed (see Note below).
        If not LISTP[macval] or if CDR[macval] = NIL:
          Return READ[file;rdtbl];
        elseif CAR[macval] = CDR[macval],
          return CAR[CAR[macval]];
        else, return CAR[macval].
    else:
      Let charlst be the proper list of Characters corresponding to
      the sequence of characters obtained by reading at least one
      character and until the next character to be read is a break
      or separator character.
      Return PACK[charlst].
```

Note: If the body of an armed read macro attempts to read a RIGHTPAREN or RIGHTBRACKET (with a call to READ in or under the read macro body) while that call is not assembling a List Structure (at some level), the character should not be removed from the file (or line buffer) and error 37 with culprit NIL should be caused. If the call is assembling a List Structure and the character to be read is a RIGHTBRACKET, it should be read (i.e., used as the result of the read procedure) but not removed from the file (or line buffer) unless it was matched by a LEFTBRACKET read by the internal call. This allows the RIGHTBRACKET to close LEFTPAREN characters read both by calls to READ inside and outside the body.

Below are three examples. If the top-level call to READ is presented with the character sequence '(A B $ C)' where '$' is a read macro then a call to READ within the body of '$' may read the 'C'. However, if a second call to READ in or under the body of '$' attempts to read the ')' an error is generated. If the top-level call to READ is presented with '(A B $ (C))' the body of '$' is permitted to read the '(C)' with an inner READ, but an error would occur if an attempt to read the second ')' was made, since no List Structure was being assembled in the inner READ. Finally, if '(A B $ (C]' is presented, the call to READ in '$' can read the ']' since it is assembling a List Structure, but the ']' should not be removed from the line buffer so that the top-level call to READ will still see it.

```
SETREADMACROFLG[flg]
                If there is a frame extension in the clink chain
                of *actframe* which is marked as either an armed or
                unarmed call to a read macro:
                  Let frame be the first such frame.
                  If frame is marked as armed, let oldflg be T;
                  else, let oldflg be NIL.
                  If flg, mark frame as armed;
                  else, mark frame as unarmed.
                  Return oldflg;
```

else, return NIL.

INREADMACROP[] If there is a frame extension in the clink
chain of *actframe* which is marked as an armed
call to a read macro:
Represent and return as an Integer
the number of List Structures
being assembled by the various recursive
calls to READ under the top-level call to
READ under which the read macro is being evaluated.
else, return NIL.

SKREAD[file;rereadstr]
Check File Name file for input.
Let ptr be the file pointer of file.
If rereadstr=NIL, let rereadstr be the empty String.
Let n be the number of characters in the pname of rereadstr.

Let newptr be the value that would be found in the file pointer
field of file if the following hypothetical situation were
the case and READ[file;ORIG] had just been performed:
The n characters in file preceding the one addressed by
ptr were those of the pname of rereadstr and the file
pointer of file were positioned at the first character in
this hypothetical occurrence of rereadstr.

If newptr > ptr, set the file pointer of file to newptr.

If the hypothetical READ would have immediately encountered
a ')' character,
return the Character ')';
elseif this hypothetical READ would have encountered
any unmatched right paretheses (or brackets),
return the Character ']';
else, return NIL.

READC[file] Check File Name file for input.
Read and return the next Character from file
(filling until T).

PEEKC[file;flg] Check File Name file for input.
If flg, then in the following read operation
proceed as though the control field of the
Terminal Table in use contained T.
Let char be the next Character to be
read from file (filling until T) but do
not remove the character from the file (or line buffer).
Return char.

RSTRING[file;rdtbl]
Check File Name file for input.
Check Read Table rdtbl and use rdtbl implicitly below.
Observe the ESCAPE guideline.
Let seq be the sequence of characters obtained by reading from
file (filling until a break or separator character is fetched)
until the next character to be read is a break or separator
character.
(Note that seq may be the empty sequence.)
Create and return a new String with seq
as its pname.

RATOM[file;rdtbl]
Check File Name file for input and use file implicitly below.
Check Read Table rdtbl and use rdtbl implicitly below.
Observe the ESCAPE guideline.
In the following, all filling operations are to be
done until either the first break character is
fetched, or until the first separator character
following a non-separator character is fetched.

If the next character to be read is a separator
character, read characters until the next character
to be read is a non-separator character.

Let charlst be the proper list of Characters
corresponding to the sequence of characters obtained
by reading one character and then continuing
reading until the next character to be read
is a break or separator character.
Return PACK[charlst].

LASTC[file] Check File Name file for input.
If no character has been read from file,
return ome (unspecified) Character;
else, return the last Character read from file.

RATEST[flg] Let seq be the sequence of characters parsed
by the last call to RATOM or READ (whichever
was most recently executed).
If seq was not parsed into an Atom (i.e., READ
was the last called and it did not return
an Atom):
Except for the requirement that no error
be caused, RATEST is unspecified in this
situation;
elseif flg = T:
If seq was preceded by a separator character,
return T;
else return NIL;
elseif flg = NIL:
If seq consisted of a single break character,
return T;
else return NIL;
elseif flg = 1:
If seq contained an ESCAPE,
return T;
else return NIL.

The following three functions allow the user to manipulate the contents of the line buffer and the system input buffer. We assume the existence of two additional buffers, used by CLEARBUF to hold characters removed from the two standard buffers.

Definition: The "LINBUF-buffer" is a buffer of the same length as the line buffer. The "SYSBUF-buffer" is a buffer of the same length as the system input buffer. These four buffers and the two interrupt buffers are all distinct.

BUFP[] If the line buffer is empty, return NIL;
else, represent and return as an Integer
the number of characters currently in the
line buffer (the contents of its deposit pointer).

```
READP[file:flg] Check File Name file for input.
                If file=T:
                  If BUFP[]:
                    If flg, return T;
                    elseif PEEKC[T;T] is the EOL Charater in
                    the primary Terminal Table, return NIL;
                    else, return T;
                  else, return NIL;
                elseif the file pointer of file is less than
                the end of file pointer for file:
                  If flg, return T;
                  elseif PEEKC[file] is the EOL Character in the
                  primary Terminal Table, return NIL;
                  else, return T;
                else, return NIL.

CLEARBUF[flg]   If flg:
                  If the line buffer and the system input buffer
                  are both empty, return NIL;
                  else:
                    Copy the line buffer into the LINBUF-buffer.
                    Copy the system buffer into the SYSBUF-buffer.
                    Return NIL;
                else:
                  Clear the line buffer.
                  Clear the system input buffer.
                  Return NIL.

LINBUF[flg]     If flg:
                  If the LINBUF-buffer is empty, return NIL;
                  else, create and return a new String
                  representing the character sequence corresponding
                  to the LINBUF-buffer;
                else:
                  Clear the LINBUF-buffer.
                  Return NIL.

SYSBUF[flg]     (Same specification as LINBUF except that
                "SYSBUF-buffer" is used instead of "LINBUF-buffer".)

BKLINBUF[str]   If STRINGP[str]:
                  Clear the line buffer.
                  For every successive character, char, in str
                  (or until the line buffer is replete), deposit char in
                  the line buffer.
                  Return str.

BKSYSBUF[str]   (Same specification as BKLINBUF except that
                "system input buffer"is used instead of
                "line buffer".)

FILEPOS[pat;file;start;end;skip;tail]
                Check File Name file for input.
                If start=NIL, let start be GETFILEPTR[file].
                if end=NIL, let end be GETEOFPTR[file].

                If not FIXP[start], let start be FIX[start].
                If not FIXP[end], let end be FIX[end].
                If either start or end is less than 0 or greater
                than GETEOFPTR[file], cause error 17 with
                culprit CONS["Attempt to read past end of file"].
```

```
                    If there is an integer, i, start =< i < end,
                    such that the pname of pat and the patlen long
                    character sequence containing the characters in file
                    starting with the ith are equal with respect to the
                    wild card skip (cf. Section 12):
                      Let i be the smallest value denoted by such an i.
                      If tail, let newptr be the representation as an Integer
                      of the integer i+patlen;
                      else, let newptr be the representation as an Integer
                      of the integer i.
                      SETFILEPTR[file;newptr].
                      Return newptr;
                    else, return NIL.

COPYBYTES[infile;outfile;start;end]
                    Check File Name infile for input.
                    Check File Name outfile for output.
                    If not FIXP[start], let start be FIX[start].
                    If not FIXP[end], let end be FIX[end].

                    SETFILEPTR[infile;start].
                    Let bytecount be end-start.
                    If bytecount<0, cause error 17 with culprit
                    CONS["Negative number of bytes to copy";bytecount].

                    For i from 1 to bytecount do the following:
                      Read the next character, char, from file infile.
                      Write character char to file outfile.

                    Return T.
```

28. STORAGE ALLOCATION

As noted in Section 2, INTERLISP programs can dynamically create "new" objects using "creation functions" supplied in the VM. An object is considered "new" if it is EQ to no object the user could obtain before invoking the creation function. It is desirable to allow the creation of an arbitrarily large number of objects. But of course, since it takes a certain non-zero amount of storage to represent an object, and since there is (presumably) only a finite amount of storage available, one can only represent a finite number of objects at any one time. However, most of the time the user cannot obtain all of the objects he has created, simply because he has discarded all of the references to some of them. Thus, the implementor is free to collect these "unreachable" objects and reuse the storage associated with them. This process is called "garbage collection". If at any given time the user happens to be able to reference no more objects than can be represented at once, garbage collection provides an illusion of infinite storage.

The VM does not require the existence of a garbage collector. (However, the utility of an implementation without a garbage collector will suffer greatly unless enormous amounts of storage are available.) Whether or not a garbage collector is present it is still possible to exhaust the amount of physical space available for the representation of objects. This document does not specify the action taken by the VM when it cannot fullfill a request for the creation of a new object due to lack of space. However, that action must make it clear to the user that this situation has arisen (rather than, say, merely begin reusing valid objects).

If a garbage collector is present, the VM puts very few constraints on its behavior.

The garbage collector may be invoked automatically at any time. We make the convention that every garbage collection is initiated in order to reclaim space for the representation of a particular data type. This is called the "type" of that activation of the garbage collector. Garbage collection may alter the state of the actual machine in any way the implementor desires, so long as the following condition holds:

> If the garbage collection message is NIL and the garbage collection trap field contains -1 (see below), it must not be possible for any INTERLISP program, using VM functions other than GCGAG, GCTRP, RECLAIM, STORAGE, CLOCK and DATE to detect whether or not a garbage collection has occurred, with the single exception that the program may abort or give warning messages due to lack of storage if garbage collections are avoided.

The VM requires the existence of two fields, used to provide a limited amount of user access to the garbage collector:

(1) The "garbage collection message" field, which contains some object.

(2) The "garbage collection trap" field, which contains an integer.

The use of these fields is as follows:

If the garbage collection message field contains T, the implementor should print (to the terminal) some informative message on entry to and on exit from the garbage collector[14]. If the garbage collection print flag is NIL, no message is printed on entry or exit. If the garbage collection message field contains a String, str, then PRIN1[str;T] is executed on entry to the garbage collector, and no message is printed on exit. If the garbage collection message is some List Cell (m_1:m_2), then PRIN1[m_1;T] is executed on entry to the garbage collector, and PRIN1[m_2;T] is executed on exit from the garbage collector. The action taken when the garbage collection message field is other than NIL, T, a String or a List Cell is left to the implementor.

If the contents of the garbage collection trap field is some integer, n, and at any time the total number of new List Cells which could be represented equals n, then at the next safe function call (cf. Section 25) (of some function fn with argument list args), INTERRUPT[fn;args;3] should be executed.

Initially, the garbage collection message field shall contain T and the garbage collection trap field shall contain -1.

```
GCGAG[mess]     Let oldmess be the contents of the
                garbage collection message field.
                Set the garbage collection message field to mess.
                Return oldmess.

GCTRP[n]        Let oldgctrpn be the contents of the
```

---

14 In INTERLISP-10, the entry message is simply "GC: " followed by the type of the garbage collection. The exit message says how many words of that type of storage were actually reclaimed, and how many words remain.

```
                garbage collection trap field.
                Let n be FIX[n].
                Set the garbage collection trap field to
                the integer represented by n.
                Represent and return the Integer representing oldgctrpn.

RECLAIM[type]   Initiate a garbage collection of type type.
                The implementor may define (and document)
                the result returned by RECLAIM[15].
```

Note: If no garbage collector is present, this function would be a no-op.

```
STORAGE[]       Print any information deemed by the implementor
                to be useful to the user who wishes to ascertain
                the kinds and amounts of storage currently
                in use (or allocated) to the VM.
                Return NIL.
```

29. MISCELLANEOUS VM FUNCTIONS

Definition: The "VM ordering" is a partial order on the universe of VM objects, such that Numbers (both Integers and Floating Point Numbers) are less than Literal Atoms and Strings, Literal Atoms and Strings are less than List Cells, and List Cells are less than all other objects. Within these constraints, Numbers (both Integers and Floating Point Numbers) are ordered according to signed magnitude and Literal Atoms and Strings are ordered alphabetically according to pname (the ordering of the characters of the alphabet being that of the character codes).

```
ALPHORDER[x;y]  If x is less than y in the VM ordering,return T;
                else, return NIL.

COPYALL[x]      If LISTP[x]:
                  return CONS[COPYALL[CAR[x]];COPYALL[CDR[x]]];
                if LITATOM[x], return x;
                elseif FIXP[x], represent and return as an Integer the
                integer represented by x;
                elseif FLOATP[x], represent and return as a Floating
                Point Number the real represented by x;
                elseif STRINGP[x], return CONCAT[x];
                elseif ARRAYP[x]:
                  Let size be ARRAYSIZE[x].
                  Let typ be ARRAYTYP[x].
                  If typ=FIXP:
                    Create and return a new Array of size size
                    and type typ, containing in its successive
                    fields the same succession of unboxed Integers
                    as in x;
                  else (typ=POINTER):
```

---

15 In INTERLISP-10, the result is the total number of words available for storage of data of type type, after the garbage collection.

Create and return a new Array of size size and type POINTER, such that the ith field, 1=<i=<size, contains COPYALL[ELT[x;i]];
elseif HARRAYP[x]::
Create and return a new Hash Array, newx, of the same size as x, such that for every hash-link in x, with hash-item item and hash-value val, newx contains a new hash-link with hash-item COPYALL[item] and hash-value COPYALL[val] and no other hash-links;
elseif x is a User Data Type:
Create and return a new object, newx, of the same type as x, such that for every field in x which contains some object, obj, the corresponding field in newx contains COPYALL[obj], and for every field in x which contains some meta-object, the corresponding field in newx contains the same meta-object;
elseif STACKP[x], create and return a new Stack Pointer containing the frame extension in x;
elseif READTABLEP[x], return COPYREADTABLE[x];
elseif TERMTABLEP[x], return COPYTERMTABLE[x];
else, return x;

The following two functions assume the existence of a clock, which can be used to measure both elapsed real time and elapsed time spent in computing (rather than i/o waits).

CLOCK[n]
If EQP[n;0]:
Represent and return as an Integer the number of milliseconds which have elapsed since the clock was initialized;
elseif EQP[n;1]:
Represent and return as an Integer the number of milliseconds which elapsed between the time the clock was initialized and the time the VM was entered;
elseif EQP[n;2]:
Represent and return as an Integer the number of milliseconds of compute time spent in the VM;
elseif EQP[n;3]:
Represent and return as an Integer the number of milliseconds the VM has spent in garbage collection (if a garbage collector is present).

Note: If some of these quanities cannot be computed the implementor is responsible for documenting this.

DISMISS[n]
If not FIXP[n], let n be FIX[n].
Wait n milliseconds and return NIL.

Definition: The "VM format for a date and time" is a character sequence giving a day of the month (as an integer) dy, the name of a month (or an abbreviation), mo, a year (or the last two decimal digits), yr, and an elapsed time since midnight, measured in hours, hr, minutes, mi, and seconds, sc, in the format: dy-mo-yr hr:mi:sc.

DATE[]
Create and return a new String whose pname denotes the current date and time in the VM format.

IDATE[x]
If the pname of x represents a date and time in the VM format:
Represent and return as an Integer some integer

i, such that for all objects, y, whose pnames represent a date and time in the VM format, i = IDATE[y] if and only if x and y represent the same date and time, and i < IDATE[y] if and only if the date and time represented by x occurs chronologically before that represented by y.

USERNAME[] Create and return a new String whose pname is the name of the user.

SYSOUT[file] Let file be OPENFILE[file;OUTPUT;NEW;bytesize], where bytesize is an implementation defined Integer. Write sufficient information to file so as to allow the function SYSIN (below) to completely reconstruct the state of the Virtual Machine as of the completion of this statement (with the exception of certain externally controlled features such as the real-time clock or open files, all of which should be documented).
CLOSEF[file].
Return file.

SYSIN[file] Let file be OPENFILE[file;INPUT;OLD;bytesize], where bytesize is an implementation defined Integer. Assuming file is a file constructed by SYSOUT, reconstruct the state of the Virtual Machine at the time the SYSOUT occurred.
CLOSEF[file].
Return LIST[file] (this will return from what (at the time of the SYSOUT) was the call to SYSOUT).

LOGOUT[] Exit the Virtual Machine and reenter the host operating system.

ACKNOWLEDGEMENTS

This document could not have been written without the support of Warren Teitelman, Peter Deutsch, and Larry Masinter, all of the Xerox Palo Alto Research Center, and the support of Alice Hartley and Daryle Lewis, both of Bolt, Beranek and Newman. I also wish to thank Carol Van Jepmond for preparing the final version of the manuscript. If this document helps others implement INTERLISP, we will all consider our efforts adequately rewarded.

REFERENCES

[1] Bobrow, D. G., and Wegbreit, B. "A Model and Stack Implementation for Multitple Environments", *Communications of the ACM*, Vol. 10. 10, October, 1973.

[2] Teitelman, W. INTERLISP Reference Manual, Xerox Palo Alto Research Center, 1974.

INDEX