CARNEGIE-MELLON UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

SPICE PROJECT

Internal Design of Spice Lisp

Scott E. Fahlman
Guy L. Steele
Gail E. Kaiser[1]
Walter van Roggen

20 December 1981

Spice Document S026
Keywords and index categories: PE Lisp
Location of machine-readable file: SLGUTS.MSS.24 @ CMU-20C

Supported by the Defense Advanced Research Projects Agency, Department of Defense, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under contract F33615-78-C-1551. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the ~~U.S. Government.~~

[1]Gail E. Kaiser is supported by the Fannie and John Hertz Foundation.

Table of Contents

1. About This Document

1.1. Scope and Purpose

This document describes the internal details, formats, and mechanisms of Spice Lisp, as it is implemented on microcodable personal machines. It is intended to be the definitive document on these things: if the code does not do what is described here (that is, in the *current* version of this document), it is considered to be a bug until it is proven not to be and the document is changed. This is a working document, and it will change frequently as the system is developed and maintained. All questions and comments on this material should be directed to Scott Fahlman @ CMUA.

Spice Lisp is described here for 32-bit microcodable machines, or machines which can easily be viewed as having 32 bits per word. Where some detail refers specifically to the Perq implementation, we have tried to indicate this. One could obviously make a version of Spice Lisp that would run on a 36-bit machine, or one with some other word size, by adjusting the lengths of various fields, but we will not try to indicate here what these adjustments would be. Implementations of Common Lisp also exist for conventional machines with fixed instruction sets; these are very different internally and are described in a other documents.

1.2. Bit and Byte Numbering

All addresses within Spice Lisp refer to 32-bit words. The low-order bit of each word is numbered 0; the high-order bit is numbered 31. If a word is broken into smaller units, these are packed into the word from right to left. For example, if we break a word into bytes, byte 0 would occupy bits 0-7, byte 1 would occupy 8-15, byte 2 would occupy 16-23, and byte 3 would occupy 24-31. Similarly, on the 16-bit PERQ, a word-pointer P would point to a word whose low-order half is at PERQ address 2P and whose high-order half is at 2P+1. In these conventions we follow the conventions of the VAX; the PDP-10 family follows the opposite convention, packing and numbering left to right.

All Spice Lisp documentation will use decimal as the default radix; other radices will be indicated by a subscript (as in 77_8) or by a clear statement of what radix is in use.

1.3. A Note About NIL

Spice Lisp uses a special value, written "()", to indicate both the empty list and the null or false value returned by predicates. This value is distinct from the symbol whose name is "NIL". It will be referred to in this document as "the empty list" or as "the null value" or simply as ().

[illegible]

The null value () evaluates to itself. CAR and CDR can be performed on this value, returning () in either case. () is considered to be an ATOM but not a SYMBOL; it is a LIST, but not a CONS.

NIL is a normal symbol just like any other, except the its value is permanently set to (). It is an error to try to setq or bind NIL. The control structures such as AND and NULL test for (), not NIL. Since NIL always evaluates to (), imported code that uses constructs such as (RETURN NIL) should work as expected; only code that uses 'NIL or the equivalent should cause any problems.

The value of the symbol T is permanently bound to T, and this value is the one returned by predicates that need some arbitrary non-null value.

2. Data Formats for Spice Lisp

2.1. Lisp Object Data Formats

Lisp objects are 32 bits long. They come in 16 basic types, divided into three classes: immediate data types, pointer types, and forwarding pointer types. The storage formats are as follows:

Immediate Data Types:

```
-------------------------------------------------------------------------
| Type Code (4) |                 Immediate Data (28)                   |
-------------------------------------------------------------------------
```

Pointer and Forwarding Types:

```
-------------------------------------------------------------------------
| Type Code (4) | Space Code (2) |     Pointer (24)        | Unused (2) |
-------------------------------------------------------------------------
```

2.2. Table of Type Codes

Code	Type	Class	Explanation
0	Misc	Immediate	Various distinguished values.
1	Fixnum	Immediate	28-bit integer.
2	Short-Flonum	Immediate	28-bit floating point.
3	U-Vector	Pointer	=> Vector of unboxed data words.
4	String	Pointer	=> Vector of unboxed bytes.
5	Xnum	Pointer	=> A bignum or long flonum.
6	Ynum	Pointer	=> A ratio or complex number.
7	B-Vector	Pointer	=> Vector of boxed cells.
8	Function	Pointer	=> Compiled code for 1 function.
9	Array	Pointer	=> Array header, like a B-Vector.
10	Symbol	Pointer	=> Lisp symbol, 6 boxed cells.
11	List	Pointer	=> List cell, 2 boxed cells.
12	Unused		
13	Unused		
14	EVC-Forward	Forward	==> External value cell, list space.
15	GC-Forward	Forward	==> Object in newspace of same type.

2.3. Table of Space Codes

0	Dynamic-0	Storage normally garbage collected, space 0.
1	Dynamic-1	Storage normally garbage collected, space 1.
2	Static	Permanent objects, never moved or reclaimed.
3	Read-Only	Objects never moved, reclaimed, or altered.

2.4. Immediate Data Type Descriptions

Fixnum A 28-bit two's complement integer.

Short-Flonum Short format floating point, allocated as follows:

```
 31           28      27        26        19   18                 0
------------------------------------------------------------------
| Type code (4) | Sign (1) |  Expt (8)  |  Mantissa (19)  |
------------------------------------------------------------------
```

The sign of the mantissa is moved to the left so that these flonums can be compared just like fixnums. The exponent is the binary two's complement exponent of the number, plus 128. The mantissa is a 21-bit two's complement number with the sign moved to bit 27 and the leading significant bit (which is always the complement of the sign bit and hence carries no information) stripped off. This encoding gives a range of about 10^{-38} to 10^{+38} and about 6 digits of accuracy. Note that long-flonums are available for those wanting more accuracy, but they are slower to use because they generate garbage that must be collected later.

Misc Reserved for assorted special values and semi-internal values. The 28-bit data field is divided into 4 bits of sub-type and 24 bits of data. So far, the following sub-types have been defined:

0 Trap Illegal object trap. If you fetch one of these, it's an error except under very specialized conditions. Note that a word of all zeros is of this type, so this is useful for trapping references to unititialized memory. This value also is used in symbols to flag an undefined value or definition.

1 Character A character object whose low-order 24 bits contain the following information: bits 0-7, the character code; bits 8-15, reserved for various control bits; bits 16-23, the font code. For further details see "A Character Standard for Lisp" by Guy Steele, to which Spice Lisp adheres. [illegible]

2 System-area-ptr The low 24 bits are a pointer into the system table area. Pointers of this type are used in remote U-Vectors, and perhaps in other places. Objects in this area do not move around and are not allocated by the user.

3 Control-stack-ptr
The low 24 bits are a pointer into the control stack, relative to its base. Pointers of this form are returned by certain system routines for use by debugging programs.

4 Binding-stack-ptr
The low 24 bits are a pointer into the binding stack, relative to its base. Pointers of this form are returned by certain system routines for use by debugging programs.

5 The () value The low 24 bits are always 0.

6 Values-marker Used to mark the presence of multiple values on the stack. The low 24 bits indicate how many values are being returned. These are pushed in order, then the Values-Marker is pushed.

7 Closure-marker Goes in the CAR of a CLOSURE in list space. The low 24 bits are 0.

8 Frame-Header Marks the start of each frame on the control stack. The low-order 24 bits are used for certain special kinds of calls.

For a normal function call, as created by the CALL or CALL-0 instruction, the low 24 bits are always 0.

Bit 23, if 1, indicates a "break" type call frame: when the function returns, the returned value is discarded instead of being pushed on the stack and the condition code is restored to its value before the call. Bits 0, 1, and 2 contain the saved NULL, ATOM, and ZERO indicators, respectively.

Bit 22, if 1, indicates an "escape to macro" call frame, created when a macro-instruction cannot be completed entirely within the microcode. In this case, bits 16-17 indicate where the result goes, as specified by the A field of the instruction that bailed out; bits 0-15 contain a 16-bit offset for A modes that need an offset. See section 6.9 for details. [illegible]

Bit 21, if 1, indicates a call frame that expects multiple values to be returned. Such frames are created by Call-Multiple. On return, any non-negative number of values (including 0) may have been pushed on the stack, followed by N, the number that actually were pushed.

Bits 23, 22, and 21 are mutually exclusive. It is undefined what happens when more than one of these are on at once.

Bit 20, if 1, indicates a call associated with %SP-CATCH or some other catch function. The first argument to this function must be the catch tag. This bit is set by the MARK-CATCH-FRAME misc-op. See 6.11 for details.

9 Unsupplied-Arg
Placed in stack frame argument slots to indicate optional arguments that were not passed by the caller and must be given default values. The low 24 bits are 0.

10 Frame-Barrier Placed at the end of the args-and-locals area of an active control stack frame. Use by stack-examining functions to find their way around. [In the Perq implementation, this also ensures that the current args-and-locals area will not extend into the TOS register, which would make

[illegible]

extra work for the microcode. See section 6.1 for a description of the TOS register.] The low 24 bits are 0.

2.5. Pointer-Type Objects and Spaces

Each of the pointer-type lisp objects points into a different space in virtual memory. There are separate spaces for Xnums, Code, Symbols, Lists, and so on. The 4-bit type-code provides the high-order virtual address bits for the object, folllowed by the 2-bit space code, followed by the 24-bit pointer address. This gives a 30-bit virtual address to a 32-bit word; on a byte-addressed machine, the two low-order bits will be 0. In effect we have carved a 30-bit space into a fixed set of 24-bit subspaces, not all of which are used. In a machine with a virtual address space smaller than 30 bits, some method of folding these spaces into the smaller space would have to be developed.

[Note: On the PERQ there are significant performance advantages to be gained by aligning all objects on the PERQ's "quad-word" (64-bit) boundaries. This happens automatically for LIST objects, which are two LISP-words long and for symbols, which are 6 long. For all other pointer-type objects, the allocator makes sure that the object starts on a quad-word boundary, wasting a word with a MISC-TRAP code if necessary. Thus, on the PERQ, every 24-bit pointer (except for control stack pointers) will have a low-order bit of 0. This is a peculiarity of the PERQ version of Spice Lisp and is not part of the specification of Spice Lisp itself. User-level code should never have to notice this distinction.]

The space code divides each of the type spaces into four sub-spaces, as shown in the table above. At any given time, one of the dynamic spaces is considered newspace, while the other is oldspace. The garbage collector continuously moves accessible objects from oldspace into newspace. When oldspace contains no more accessible objects it is considered empty. A "flip" can then be done, turning the old newspace into the new oldspace. All type-spaces are flipped at once. Allocation of new objects always occurs in newspace.

Optionally, the user (or system functions) may allocate objects in static or read-only space. Such objects are never reclaimed once they are allocated -- they occupy the space in which they were initially allocated for the lifetime of the Lisp process. The advantage of static allocation is that the GC never has to move these objects, thereby saving a significant amount of work, especially if the objects are large. Objects in read-only space are static, in that they are never moved or reclaimed; in addition, they cannot be altered once they are set up. Pointers in read-only space may only point to read-only or static space, never to dynamic space. This saves even more work, since read-only space does not need to be scavenged, and pages of read-only material do not need to be written back onto the disk during paging.

A type-space will contain boxed objects or unboxed objects, but not both. Similarly, a space will contain either fixed-length objects or variable-length objects, but not both. A variable-length object always contains a 24-bit length field right-justified in the first word, with the Fixnum type-code in the high-order four bits. The remaining four bits can be used for sub-type information. The length field gives the size of the object in 32-bit words, including the header word. The garbage collector needs this information when the object is moved, and it is also useful for bounds checking.

The format of objects in each space are as follows:

Symbol

Each symbol (a non-numeric Lisp atom) is represented as a fixed-length block of boxed Lisp cells. The number of cells per symbol is 6, in the following order:

```
0  Value cell for shallow binding.
1  Definition cell: points to function or list space.
2  Property list: a list of attribute-value pairs.
3  Print name: points to a string.
4  Package: points to the obarray holding this symbol.
5  Hash: A fixnum, the SXHASH value for this symbol.
```

List

A fixed-length block of two boxed Lisp cells, the CAR and the CDR.

B-Vector

Vector of boxed 32-bit lisp objects, any length. The first word is a fixnum giving the number of words allocated for the vector (up to 24 bits). The highest legal index is this number minus 2. The second word is vector entry 0, and additional entries are allocated contiguously in virtual memory. (See section 3 for further details.)

U-Vector

Vectors of unboxed data items, any length. The 24 low bits of the first word give the allocated length in 32-bit words. The low-order 28 bits of the second word gives the length of the vector in entries, whatever the length of the individual entries may be. The high-order 4 bits of the second word contain access-type information that yields, among other things, the number of bits per entry. Entry 0 is right-justified in the third word of the vector. Bits per entry will normally be powers of 2, so they will fit neatly into 32-bit words, but if necessary some empty space may be left at the high-order end of each word. (See section 3 for details.)

Xnum

Xnums are unboxed and of variable length, and are identical in format to U-Vectors. Because of this, bignums can be accessed and altered by the same macro-instructions that are used for U-Vectors. The first word of each Xnum contains the fixnum type-code in the first 4 bits, a sub-type code in the next 4 bits, and a 24-bit length field; the second word contains an access-type code and a number of entries. Currently there are two sub-types of Xnums: bignums and long flonums.

Code avail?

Bignums are infinite-precision integers. We envision that some implementations of Spice Lisp will implement bignum arithmetic in macrocode and some in microcode. On the current PERQ, all bignum arithmetic is in macrocode. Each bignum is stored as a series of 8-bit bytes, with the low-order byte stored first. The representation is two's complement, but the sign of the number is redundantly encoded in the subtype field of the bignum:

positive bignums are sub-type 0, negative bignums sub-type 1. The access-type code is always 8-Bit.

Long flonums have sub-type code 2, and always contain 3 words of data plus the 2-word header. The first data word is the 32-bit exponent in two's complement form. This is followed by two words (64 bits) of mantissa, also in two's complement form, with the low-order word first. This gives about 19 digits of accuracy and more range than you need. For long flonums we do not bother with stealing the meaningless most-significant bit, as the small amount of added accuracy will not justify the extra work. The access-type code is also 8-Bit.

Ynum A Ynum is an extended-format number like an Xnum, but the form is that of a B-Vector rather than a U-vector. Thus, a Ynum can be built from any number of boxed Lisp objects. Currently the only Ynum type defined is is the Ratio, but this format may be used in the future for non-normalized ratios, complex numbers and other esoterica.

A ratio is a Ynum of subtype 0. Ratios always contain two boxed items: the numerator (index 0) and the denominator (index 1). Both must be integers (fixnums or bignums). Ratios are normally stored in normalized form, with all common divisors factored out; if the resulting denominator is 1, the normalized form for the ratio will in fact be represented as a fixnum or bignum.

Array This is actually a header which holds the accessing information and other information about the array. The actual array contents are held in a vector (either U-vector or B-vector) pointed to by an entry in the header. The header is identical in format to a B-vector. For details on what the array header contains, see section 3.3.

String A vector of unboxed bytes. Identical in form to U-Vectors with the access type always 8-Bit. However, instead of accepting and returning fixnums, string accesses accept and return character objects (see section 5.8). Only the 8-bit code field is actually stored, and the returned character object always has bit and font values of 0.

Function A compiled Spice Lisp function consists of both boxed and unboxed information. The boxed information is stored in the Function space in a format identical to that used for boxed vectors, with a 24-bit length field in the first word. This object contains assorted parameters needed by the calling machinery, a number of pointers to symbols used as special variables within the function, and a number of lisp objects used as constants by the function. There is also a pointer to an unboxed "Code vector" that holds the actual compiled byte codes for the function. For details of the code format and definitions of the byte codes, see section 5.6.

2.6. Forwarding Pointers

GC-Forward When a data structure is transported into newspace, a GC-Forward pointer is left behind in the first word of the oldspace object. This points to the same type-space in which it is found. For example, a GC-Forward in B-vector space points to a structure in the B-vector newspace. GC-Forward pointers are only found in oldspace, and they always point into the corresponding newspace.

EVC-Forward	This is used to implement closures. (See section 6.8.) It is placed in the value cell of a symbol to indicate that the value is to be found in a cell of a closure structure in list space. Points to a list cell whose CAR is the external value cell. Always considered to point to some list space, new or old.

2.7. System and Stack Spaces

Some of the system's type-spaces are unused because they correspond to immediate data types. Each unused sub-space is 2^{24} words long. The system's stacks and control areas are hidden in these spaces. Currently the following spaces are allocated:

Type Code	Space Code	Use
0	0	System Tables.
0	1	Control Stack.
0	2	Special Binding Stack.

The system tables area is used by the Lisp system for various in-core data structures. These tables are fixed in size and location in virtual memory. They are not managed by the garbage collector. Among the things stored here are the table of garbage collection pointers for the various spaces, some I/O status tables and buffers, a table of microcode entry points, and so on. Some of the system tables structures want to be accessible from Lisp; this is accomplished by the use of the "remote" U-Vector mechanism. (See section 3.)

The control stack grows upward (toward higher addresses) in memory, and is a framed stack. It contains only boxed Lisp objects (with some random things encoded as fixnums or Misc codes). Every object pointed to by an entry on this stack is kept alive. The frame for a function call contains an area for the function's arguments, an area for local variables, a pointer to the caller's frame, and a pointer into the linear binding stack. The stack pointers are Misc codes. The precise stack frame format can be found in a later section of this document.

The special binding stack also grows upward. This stack is used to hold previous values of special variables that have been bound. It grows and shrinks with the depth of the binding environment, as reflected in the control stack. This stack contains symbol-value pairs, with only boxed Lisp objects present.

2.8. Symbols Known to the Microcode

A large number of symbols will be pre-defined when a Spice Lisp system is fired up. A few of these are so fundamental to the operation of the system that their names have to be assembled into the microcode. These symbols are listed here. All of these symbols are in static space, so they will not be moving around.

NIL	58000000_{16} The value of NIL is always (); it is an error to try to alter it.
T	58000018_{16} The value of T is always T; it is an error to try to alter it.
ALLOCATION-SPACE	58000030_{16} This symbol's value must be fixnum 0, 2, or 3, indicating that new cons cells or other structures are to be allocated in Dynamic, Static, or Read-Only space, respectively.
%SP-INTERNAL-APPLY	58000048_{16} The function stored in the definition cell of this symbol is used by the microcode whenever compiled code calls an interpreted function. See section 6.5 for details.

3. Vectors and Arrays

Spice Lisp provides both boxed and unboxed vectors as fundamental data types, and attempts to make access to them as fast as is possible on the machine at hand. Arrays of any number of dimensions are also provided, but of course access to these is slower since index arithmetic must be done. Array access goes through a header structure, itself in the form of a boxed vector, which contains the information needed to access the array contents. The array data is actually stored in a vector, B or U as the case may be. It is possible to have two array header structures point at the same data vector, which will result in possibly different modes of accessing the same data. For example, one header might specify a 2-D array of bits with dimension N by 8, while another header might access the same data as as a 1-D array of N bytes.

3.1. B-Vectors

B-Vectors are the simplest case. B-Vectors contain only boxed lisp objects. The format is as follows:

```
---------------------------------------------------------------
| Fixnum code (4) | Subtype (4) |  Allocated length (24)      |
---------------------------------------------------------------
|  Vector entry 0   (Additional entries in subsequent words)  |
---------------------------------------------------------------
```

The first word of the vector is a header indicating its length; the remaining words hold the boxed entries of the vector, one entry per 32-bit word. The header word is of type fixnum. It contains a 4-bit subtype field, which is used to indicate several special types of B-Vectors. At present, the following subtype codes are defined:

0	Normal. Used for assorted things.
1	Named structure created by DEFSTRUCT, with type name in entry 0.
2	EQ Hash Table, last rehashed in dynamic-0 space.
3	EQ Hash Table, last rehashed in dynamic-1 space.
4	EQ Hash Table, must be rehashed.

Following the subtype is a 24-bit field indicating how many 32-bit words are allocated for this vector, including the header word. Legal indices into the vector range from zero to the number in the allocated length field minus 2, inclusive. The index is checked on every access to the vector. Entry 0 is stored in the second word of the vector, and subsequent entries follow contiguously in virtual memory.

Once a vector has been allocated, it is possible to reduce its length by the %SP-SHRINK-VECTOR operation, but never to increase its length, even back to the original size, since the space freed by the reduction may have been reclaimed. This reduction simply stores a new smaller value in the length field of the header word.

It is not an error to create a B-Vector of length 0, though it will always be an out-of-bounds error to access such an object. The maximum possible length for a B-Vector is 2^{24}-2 entries, and that is only possible if no other B-vectors are present in the space.

Objects of type FUNCTION and ARRAY are identical in format to B-Vectors, though they have their own spaces. In both cases, only 0 is currently used in the sub-type field of the header word.

Ynums are also identical in format and operation to B-Vectors, though they may also be operated on directly by microcoded routines. For details of the currently-defined sub-types and their access-codes, see section 2.5.

3.2. U-Vectors

U-Vectors contain unboxed items of data, and their format is more complex. The data items come in a variety of lengths, but are of constant length within a given U-Vector. Data going to and from a U-Vector is passed as a FIXNUM, right justified. Internally it is stored in packed form, filling 32-bit words without any type-codes or other overhead. The format is as follows:

```
------------------------------------------------------------
| Fixnum code (4) | Subtype (4) |  Allocated length (24)      |
------------------------------------------------------------
| Access type (4) | Number of entries (28)                    |
------------------------------------------------------------
|                                    Entry 0 right justified |
------------------------------------------------------------
```

The first word of a U-Vector contains the Fixnum type-code in the top 4 bits, a 4-bit subtype code in the next four bits, and the total allocated length of the vector (in 32-bit words) in the low-order 24 bits. At present, the following subtype codes are defined:

0	Normal. Used for assorted things. 1
	Code. This is the code-vector for a function object.

The second word of the U-Vector is the one that is looked at every time the U-Vector is accessed. The low-order 28 bits of this word contain the number of valid entries in the U-Vector, regardless of how long each entry is. The lowest legal index into the U-Vector is always 0; the highest legal index is one less than this number-of-entries field from the second word. These bounds are checked on every access. Once a vector is allocated, it can be reduced in size but not increased. The %SP-SHRINK-VECTOR operation changes both the allocated length field and the number-of entries field of a U-VECTOR.

The high-order 4 bits of the second word contain an access-type code which indicates how many bits are occupied by each item (and therefore how many items are packed into a 32-bit word) and also whether the access is local or remote. The encoding is as follows:

0	1-Bit	8	1-Bit-Remote
1	2-Bit	9	2-Bit-Remote
2	4-Bit	10	4-Bit-Remote
3	8-Bit	11	8-Bit-Remote
4	16-Bit	12	16-Bit-Remote
5	Unused	13	Unused
6	Unused	14	Unused
7	Unused	15	Unused

In local U-Vectors (the normal kind), the data items are packed into the third and subsequent words of the vector. Item 0 is right justified in the third word, item 1 is to its left, and so on until the allocated number of items has been accommodated. All of the currently-defined access types happen to pack neatly into 32-bit words, but if this should not be the case, some unused bits would remain at the left side of each word. No attempt will be made to split items between words to use up these odd bits.

In remote U-Vectors, only three words are allocated. The third word is a pointer (of the appropriate MISC-code subtype) into the system tables area of virtual memory. Instead of being stored within the U-Vector, the data items are stored at the system-table location pointed to. They are packed just as they would be if the word pointed to were the third word of a local U-Vector.

When allocated, a local U-Vector is initialized to all 0's. The user will not normally allocate remote U-Vectors for himself, since these are for special use by internal system code. Remote U-Vectors are often filled with non-0 data at system startup.

As with the B-Vectors, it is not an error to create a U-Vector of length 0, but it will always be an error to access such a vector. The maximum possible length of a U-Vector is 2^{28}-1 entries or 2^{24}-3 words, whichever is smaller.

Objects of type STRING are identical in format to U-Vectors, though they have their own space. Strings always have subtype 0 and access-type 3 (8-Bit). Strings differ from normal U-Vectors in that the accessing functions accept and return objects of type Misc-Character rather than fixnums.

Xnums are also identical in format and operation to U-Vectors, though they may also be operated on directly by microcoded routines. For details of the currently-defined sub-types and their access-codes, see section 2.5.

3.3. Arrays

An array header is identical in form to a B-Vector. Like any B-Vector, its first word contains a FIXNUM type-code, a 4-bit subtype code, and a 24-bit total length field (this is the length of the array header, not of the vector that holds the data). At present, the subtype code is always 0. The entries in the header-vector are interpreted as follows:

0 Type-code — This is a fixnum which contains a variety of information about how the array is to be accessed and interpreted. See the array-type-code table below. If the array contains unboxed items, one part of this type-code is an access-type field identical in form to the access-type field defined for U-Vectors. This field may be used instead of the access-type field of the data-vector in making accesses to the data. This allows for accessing a single data vector through several array headers, each with a different format.

1 Data Vector — This is a pointer to the U-Vector or B-Vector that contains the actual data of the array. In a multi-dimensional array, the supplied indices are converted into a single 1-D index which is used to access the data vector in the usual way.

2 Number of Dimensions — This is a fixnum in the range 0 to 2^{24}-6. Of course, if an array of the maximum dimensionality is created, there will probably not be enough space in virtual memory for the data. An array of 0 dimensions is legal. It creates a single storage item which can be accessed and stored by the unique 0-D index list, (). (Explanation: mathematicians define the product of 0 things as 1. APL handles 0-D arrays in this way, so we may as well do it this way too.) A 1-D array is perfectly legal; it differs from a vector in that all accesses go through the array header structure, and are therefore somewhat less efficient.

3 Number of Elements — This is a fixnum indicating the number of elements for which there is space in the data vector.

4 Fill Pointer — This is a fixnum indicating how many elements of the data vector are actually considered to be in use. Normally this is initialized to the same value as the Number of Elements field, but in some array applications it will be given a smaller value. Any access beyond the fill pointer is illegal.

5 Displacement — This fixnum value is added to the final code-vector index after the index arithemtic is done but before the access occurs. Used for mapping a portion of one array into another. If null (the default), the displacement is 0.

6 Array Leader — This is an optional B-Vector which is used to emulate the array leader construct of the Lisp

Machine. In this capacity, it stores an assortment of user-supplied information about the array or what it stands for.

7 Range of First Index
This is the number of index values along the first dimension, or one greater than the largest legal value of this index (since the arrays are always zero-based). A fixnum in the range 0 to 2^{24}-1. If any of the indices has a range of 0, the array is legal but will contain no data and accesses to it will always be out of range. In a 0-dimension array, this entry will not be present.

8 - N Ranges of Subsequent Dimensions

The ranges of all indices are checked on every access, during the conversion to a single data-vector index. In this conversion, each index is added to the accumulating total, then the total is multiplied by the range of the following dimension, the next index is added in, and so on. In other words, if the data vector is scanned linearly, the last array index is the one that varies most rapidly, then the index before it, and so on. In our initial implementation, the index computation for arrays is done in macrocode; when we have a larger microstore, we will move this to microcode for greater speed.

The type-code word is divided as follows:

Bit 0	0 = Boxed Data; 1 = Unboxed Data.
Bit 1	0 = Use Vector's own access-type code. 1 = Use the code supplied here.
Bits 4 - 7	Access type code for unboxed data, used if bit 1 = 1.
Bits 5 - 27	Unused.

4. Storage Management

4.1. Allocation

New objects are allocated from the lowest unused addresses within the specified space. Each allocation call specifies how many words are wanted, and a FREE-STORAGE pointer is incremented by that amount. There is one of these FREE-STORAGE pointers for each space, and it points to the lowest free address in the space. Since most of the virtual address space may initially be set up as non-existent memory, new allocation may require that the storage manager request additional pages of storage from the Spice kernel. (It remains to be seen whether the optimal increment is one page or a group of pages.) If this request fails, perhaps because the local disk is full, an error occurs within Lisp. Because no pointer to oldspace can pass through the machine (see below) and into a newly-allocated object, new objects are guaranteed to be free of oldspace pointers and do not need to be scavenged.

In unboxed spaces, objects copied from oldspace are allocated from low memory, just as though they were new objects. Unboxed objects are not scavenged.

In boxed spaces, objects that are copied from oldspace must be scavenged for pointers back into oldspace. Therefore, these potentially "dirty" objects are kept separate from the "clean" newly-allocated objects. This is done by storing them at the high-address end of the sub-space. A COPY-SPACE pointer is started at the highest free address in a given sub-space and is moved down by the appropriate amount as space for each object is allocated. A second pointer, the CLEAN-SPACE pointer, also begins at the top of the space and is moved down by the scavenger as it works. Everything above the CLEAN-SPACE pointer is certified to be free of pointers to oldspace.

Optional, for versions of Spice Lisp on machines where microstore is ample: treat LIST space specially by scavenging CDRs first, then CARs. This leads to substantially greater locality in the newspace list structures. This implementation uses two CLEAN-SPACE pointers, one for CARs and one for CDRs. The scavenger always works on the CLEAN-CDR pointer unless it is at the COPY-SPACE pointer; in that case, it works on the CLEAN-CAR until it too catches up to COPY-SPACE. When all three pointers coincide, the area is clean.

Read-only sub-spaces have two associated pointers: FREE-STORAGE and FREEZE. New items are allocated at the low-address end of that space's free storage, and the FREE-STORAGE is updated. Then the new structure is initialized. When this process is completed, the Freeze command is given, which moves all of the FREEZE pointers up to meet their respective FREE-STORAGE pointers. Once it is below the FREEZE pointer, the new structure becomes read-only and cannot be altered again without causing an error. Also, any pointer-type items written into read-only space, even if they are above the freeze pointer, must point into read-only or static space. Placing dynamic pointers in read-only space causes an error, since these pointers would have to be altered if the object pointed to is transported.

Sometimes, for purposes of sharing, it is desirable to keep certain read-only structures on separate pages of memory. The New-Pure-Page command updates both the FREE-STORAGE and FREEZE pointers of a given space to the next page boundary, wasting any remaining space on the current page.

4.2. The Transporter

All memory references made by the Lisp system should be made to newspace, static space, or read-only space; oldspace references can be made only for the purpose of moving the thing referenced to newspace. In order to reference an object, or any words within an object, the machine must have a pointer to that object in hand. Therefore, we can enforce the newspace-only reference policy by making sure that no pointer to an oldspace object ever makes it into those registers of the machine from which references might be generated. This is called the "barrier". If an attempt is made to fetch an oldspace pointer through the barrier, the oldspace object must first be transported to newspace; a pointer is then created to this newspace object, and it is this pointer that is passed through the barrier. Forwarding pointers are also followed and collapsed as they are moved through the barrier. We speak of objects within the barrier as being "in the machine", but some machine registers involved in the transporting process must obviously be considered outside the barrier for things to work.

Suppose we fetch lisp object B into the machine (across the barrier) from address A. Since the pointer A has already passed the barrier, it does not point into oldspace. The algorithm is as follows:

1. If B is an immediate data type, or a non-forwarding pointer into static space, read-only space, or newspace, simply let B through. If it is a forwarding pointer, go to step 5.

2. Otherwise, we have just picked up a pointer to oldspace, which is not allowed to be in the machine. Fetch what B points to, and call that C.

3. If C is of type GC-Forward, it had better point into newspace. Verify this. Form a new pointer with the type-code from B and the space-code and pointer from C. This is the new B. Write it into location A and pass this new B through the barrier.

4. If C is not GC-Forward, C is the first word of a structure that must be copied into newspace. The size is either a constant function of the space (check the type-code of B) or can be obtained from the low-order 24 bits of C. Allocate the appropriate number of words in the proper copyspace and copy all the words of the structure. Replace C in oldspace with a GC-Forward pointer to the new

copy. Create a new B pointing to the new copy. Write this into address A and pass the new B through the barrier.

5. If the original B is of type EVC-Forward, perform the above steps if necessary, but where it says "pass B through the barrier", you instead pick up the contents of the cell B points to and repeat the process from step 1, using this as the new B. The EVC-Forward pointer in memory may be transported to newspace, but it is not clobbered. If the original B is GC-Forward, this is an error, since GC-Forward should not be found in newspace.

One additional twist is needed to make it possible for the system to maintain hash tables full of Lisp objects that are accessed via an EQ test. Whenever a B-Vector with subtype code 2 or 3 is transported to newspace, its subtype code is changed to 4. This code indicates to the accessing functions that the entries of the hash table must be re-hashed before any access is made, since the items are hashed under their oldspace addresses which will be changed as soon as the item is touched. The rehashing is done by the hash-table functions in macrocode; all that the microcode has to do is change the subtype code when the vector is transported.

4.3. The Scavenger

When objects are first transported into copyspace, they are dirty -- that is, they may contain pointers into oldspace. The scavenger moves word by word through each of the copyspaces, transporting each lisp object encountered by the algorithm listed above. This gradually cleans up copyspace, though in the process new objects may be copied. The CLEAN-SPACE pointer marks the boundary between space that is guaranteed clean and space that may be dirty. Eventually, as oldspace empties, the process will converge and the CLEAN-SPACE pointer will catch up with the COPY-SPACE pointer. When one space has been completely scavenged, the scavenger goes on to another space, and continues to cycle through spaces until all are clean. At this point, we can be sure that every accessible object in oldspace has been moved to newspace, and the cycle of garbage collection is complete.

Each word is scavenged individually, and the scavenging process can be interrupted between any two words. This means that we can run the scavenger incrementally, scavenging a few words at a time interleaved with regular processing. This is normally tied to the allocation of new objects: for every word of new structure that is allocated, we scavenge some number of words of copyspace. (A typical ratio is four words scavenged for each word allocated.) The user can also specify that the scavenger is to run continuously until either the current cycle is complete or the user interrupts and stops it. This option can be exercised by I/O code to run the scavenger during waits for the keyboard and other lulls in the action. In a very large lisp system, a full scavenge cycle may take hours; disk paging time dominates.

Boxed static spaces may contain pointers into dynamic space, and so must be scavenged after a flip. The

CLEAN-SPACE pointer for such a space is set equal to the FREE-STORAGE pointer after a flip, and is moved downward by the scavenger until it hits the bottom of this space.

Read-only spaces do not contain pointers to dynamic space, so scavenging is not necessary.

Flipping

A Spice Lisp system begins with only static space, read-only space, and newspace (dynamic-0 space) in use. No scavenging or transporting is done until the dynamic space is flipped. At this point, dynamic-0 becomes oldspace and dynamic-1 becomes newspace, and the system begins transporting objects out of dynamic-0. When the scavenging process is complete, another flip can occur and objects are moved back into dynamic-0. And so on. The user can either initiate flips himself (if the system is ready to be flipped) or he can have the system do this automatically. To run without garbage collection, the user simply prevents the system from flipping.

If the system is controlling the flips, it will wait until a certain number of words have been allocated, totalled over all spaces; then it performs the first flip. After that, it will be able to flip again as soon as the previous scavenge cycle has been completed or it may wait until some amount of additional allocation has been done before flipping again. The optimal strategy will have to be determined empirically, once the system is up. Initially, we will flip by hand.

When a flip occurs, the system must immediately go through all of the registers inside the barrier and transport the contents of these registers, since many would otherwise point to oldspace. This task must be completed before the system resumes normal processing. It is therefore important to keep the number of items within the barrier reasonably small. For this reason, the binding and control stacks are considered to be outside the barrier, except for the current frame on the control stack; this is inside the barrier and must be scavenged during the flip. The remainder of this stack is scavenged from the top (most recent push) toward the bottom; this is the first thing the scavenger does, but this is not done during the flip. A CLEAN-STACK pointer separates the clean part of the stack from the part not yet scavenged. After the control stack is clean, the scavenger can go on to do the binding stack, then to the other boxed spaces. When the stack is popped, the system must check to see whether the new stack frame has been completely scavenged, and must immediately clean any part of the new top frame that needs it. Any access to a dirty part of either stack must be transported normally as the data is brought through the barrier.

5. Macro Instruction Set

5.1. General

This section documents the macro-instruction set to be used for Spice Lisp. This instruction set is based on, but not identical to, that used by the MIT Lisp Machine. Some changes are necessary because some of our underlying mechanisms are different: since we have no locative pointers and no CDR coding in the current Spice Lisp, some of the instructions used on the Lisp Machine make no sense. In addition, on the PERQ and on several other machines to which we might want to port Spice Lisp in the future, there is a clear advantage in fitting the instructions into 8-bit bytes rather than 16-bit words, even if extra bytes must usually be fetched to get address fields, etc. The current Lisp Machine instruction set does not break neatly at 8-bit boundaries. Finally, some operations have been added to make up for weaknesses that have been noticed in the MIT instruction set, notably in accessing vectors and strings, which happens much more frequently than anyone expected.

The intent is that this instruction set should be a very direct mapping from the S-expression source it is derived from. There should therefore never be any temptation for users to write macrocode by hand; all of the system that is not in microcode should be written in Lisp. Since the compiler will run both in Spice Lisp and in Maclisp, we need not hand-code things even for bootstrapping.

5.2. Function Object Format

Each compiled function is represented in the machine as a Function Object. This is identical in form to a B-Vector of lisp objects, and is treated as such by the garbage collector, but it exists in a special function space. (There is no particular reason for this distinction. We may decide later to store these things in B-Vector space, if we become short on spaces or have some reason to believe that this would improve paging behavior.) Usually, the function objects and code vectors will be kept in read-only space, but nothing should depend on this; some applications may create, compile, and destroy functions often enough to make dynamic allocation of function objects worthwhile.

The function object contains a vector of header information needed by the function-calling mechanism: a pointer to the U-Vector that holds the actual code, the number of required and optional arguments, and a few other things. Following this information is a vector of symbol pointers (for symbols that are used as special variables in the code) and constants. A constant is any boxed Lisp object that is used but not altered by the function. Constants in the range of -128 to +127 can be generated within the byte code, and so do not need to be represented here as full-word constants.

Spice Lisp uses an extended arglist format with &optional and &rest arguments. This is similar to the system used in the Lisp Machine, but we will not be using the Lisp Machine's very complex argument descriptor format. Instead we will initialize optional arguments in a simple and uniform way within the code for the function. We will not support the use of random &quote arguments within the arglist: either all of the arguments are EVALed, or you use a FEXPR which evals none of them and always has a single argument. Other quoting effects can be obtained through the use of macros, if desired. See section 6.7 for more details on how optional arguments are handled.

After the one-word B-Vector header, the entries of the function object are as follows:

```
0  A fixnum with bit fields as follows:
   0  - 14: Number of symbols/constants in this fn object (0 to 32K-1).
   15 - 26: Not used.
   27:      0 => All args evaled.  1 => This is a FEXPR.
1  Pointer to the unboxed Code vector holding the macro-instructions.
2  A fixnum with bit fields as follows:
   0  -  7: The minimum legal number of args (0 to 255).
   8  - 15: The maximum number of args, not counting &rest (0 to 255).
   16 - 26: Number of local variables allocated on stack (0 to 2047).
   27:      0 => No &rest arg.     1 => One &rest arg.
3  Name of this function (a symbol).
4  Vector of argument names, in order, for debugging use.
5  Reserved for future use.
6  Entry 0 of symbol/constant area.
7  Entry 1 ... and so on.
```

5.3. Execution Environment

At any given time, the machine contains pointers to the current top of the control stack, the start of the current active frame (in which the current function is executing), and the start of the most recent open frame. In addition, there is a pointer to the current top of the special binding stack. An open frame is one which has been partially built, but which is still having arguments for it computed. When all the arguments have been computed and saved on the frame, the function is then started. This means that the stack frame is completed, becomes the current active frame, and the function is executed. At this time, special variables may be bound and the old values are saved on the binding stack. Upon return, the active frame is popped away and the result is either sent as an argument to some previously opened frame or goes to some other destination. The binding stack is popped and old values are restored.

The active frame contains pointers to the previously-active frame, to the most recent open frame, and to the point to which the binding stack will be popped on exit, among other things. Following this is a vector of storage locations for the function's arguments and local variables. Space is allocated for the maximum number of arguments that the function can take, regardless of how many are actually supplied.

In an open frame, the structure is built up to the point where the arguments are stored. Thus, as arguments are computed, they can simply be pushed on the stack. When the function is finally started, the remainder of the frame is built. The control stack is also used by the instruction set as the primary scratchpad area for random computations.

The precise details of stack-frame formats and calling conventions can be found in a later section of this document.

5.4. Implementation Note

On the PERQ we do not have any sort of fast cache for the stack. The ESTK is shallow and does not allow access to anything but the top 16 bits, and since we cannot index into registers, we cannot stash a piece of the stack there. So the stack must live in core and we must take our lumps on this; disk paging time will probably be the dominant time-waster anyway. The cache promised for the extended PERQ will help a lot on this problem, if we ever get our hands on this.

What we can do is to save the top word of the stack in a register-pair rather than in core -- a sort of one-word stack buffer. Actually, this should help a great deal, since most of the macro instructions that reference the stack look only at the top word. If an instruction pops an argument off the stack, does something to it, and pushes the result, no memory I/O needs to occur. If the net effect of the instruction is to push or pop a word on the stack, then a memory write or read is required. The register in question is designated TOS for "top of stack".

5.5. Indicators

Conceptually, the result produced by each macro-instruction is used to set a group of indicators, which can be tested by subsequent conditional jump instructions. These indicators are NULL, ATOM, and ZERO. In fact, on most machines, it is more efficient to just stash in a hidden register the result-word that should have set the indicators, and to check for null-ness, atom-ness, or zero-ness when the question comes up. In the description of instructions below, if it is unclear what the natural "result" is, we will state explicitly what value goes into the indicators; in some cases, instructions leave the indicators unchanged.

The NULL "indicator" is set only by a result of (). Note that it is not set by the symbol NIL.

The ATOM "indicator" is set if the result is not of type LIST. This indicator *is* set by a result of ().

The ZERO "indicator" is set by either a FIXNUM 0 or a FLONUM 0.0.

5.6. Macro-Instruction Formats

The majority of the macro-instructions in the spice Lisp Set are of the following form:

```
                          7                          0
                          ---------------------------------
Instruction byte:         | OP (6)              | A (2) |
                          ---------------------------------
Next byte (optional):     |           B (8)             |
                          ---------------------------------
```

Most instructions read from or write to an "effective address", and possibly also push or pop 32-bit words on the stack. When the OP field indicates that an effective address is to be read from, it is computed from the A field and (sometimes) from the subsequent byte B as follows:

A = 0	The operand is popped off the stack. Then the operation takes place, in some cases popping a second (distinct) argument off the stack and/or pushing something onto the stack. No B byte is fetched.
A = 1	The next byte is fetched and is converted (with sign extension) to a signed fixnum in the range -128 to +127. This is used as the operand.
A = 2	The next byte is fetched. If its sign bit is 0, the remaining 7 bits are used as an unsigned offset (0 - 127) into the vector of symbols and constants in the code object of the current function. If the sign bit is 1, the other 7 bits are used instead as an unsigned offset (0 - 126) into the arguments and local variables area of the currently-active stack frame. The contents of this cell are used as the operand. If the fetched byte is all ones (377 octal), the next two bytes are fetched to form a 16-bit offset. The sign bit of this extended offset controls where the operand comes from, as in the 8-bit offsets. In fetching this double offset, the low-order byte comes in first.
A = 3	The next byte (or set of bytes) is fetched and is used as an offset into the code object or stack frame, as above. But instead of being used directly, the cell addressed is supposed to contain a symbol pointer and the operand is fetched from its value cell. If the value is Misc-Trap, an UNBOUND error is signalled.

If the effective address is being used as a place to write, the following descriptions apply:

A = 0	The result is pushed on the stack.
A = 1	The result sets the indicators, then is thrown away.

A = 2	If the offset indicates a stack frame destination, the result is put there; if it points into the code object, this destination is illegal, since the code object should not be altered.
A = 3	This writes into the value cell of the symbol pointed to, forwarding the write through an EVC-Forward pointer if one is present in the value cell.

In the following listing, the effective address is called "E" and its contents are called "CE".

5.7. Instructions

Note: In the following descriptions, the number in the left margin is the 6-bit OCTAL opcode.

00 Unused

01 Call — CE must be some sort of executable function: a code object, a lambda-expression in list space, a closure, or a symbol with one of these stored in its function cell. A call block for this function is opened, and computation proceeds to gather the arguments into the call block. This is explained in more detail in the section on Control Conventions. The indicators are left unchanged.

02 Call-0 — CE must be an executable function, as above, but is a function of 0 arguments. Thus, there is no need to collect arguments. The call block is opened and activated in a single operation.

03 Call-Multiple — Just like a Call instruction, but it sets bit 21 of the frame header word to indicate that multiple values are expected. See section 6.10.

04 Call-Maybe-Multiple
If the current stack-frame header word has bit 21 set, this is identical to Call-Multiple. If not, this is identical to Call.

05 Return Return from the current function call. After the current frame is popped off the stack, CE is pushed as the result being returned. CE also sets the indicators. See section 6.6 for more details.

06 Throw — CE is the throw-tag, normally a symbol. The value to be returned, either single or multiple, is on the top of the stack. See section 6.11 for a description of how this instruction works.

07 Unused.

10 Push — CE is pushed onto the stack and sets the indicators. If A = 0, this is a NOOP, except that the indicators are set according to the value of the item on top of the stack.

11 Push-Last — CE is pushed onto the stack as the last operand for the most recent currently-open call block. The call is then activated: the call block is finished and becomes the current stack-

frame. If A = 0, the effect of this operation is just to start the call.

12 Push-Under	CE is pushed onto the stack as the second item and sets the indicators; the top item of the stack is unchanged. If A = 0, this swaps the top two items on the stack. Push-Under causes an error if the stack is empty or if A = 0 and the stack contains only one item.
13 Check	CE is used to set the indicators, but is not put anywhere. If A = 0, the net effect is to pop the stack by one word, setting indicators.
14 Pop	Pop the top item off the stack and store it in E and also in the indicators.
15 Copy	Copy the item on top of the stack into E, setting the indicators, without popping the stack.
16 Make-Predicate	If the NULL indicator is on, put a () in E. Else, put a T in E. The () or T also sets the indicators.
17 Not-Predicate	If the NULL indicator is not on, put a () in E. Else, put a T in E. The () or T also sets the indicators.
20 CAR	CE had better be a pointer into list space or (). Its CAR is pushed on the stack and sets the indicators.
21 CDR	The CDR of CE is pushed on the stack and sets the indicators.
22 CADR	The CADR of CE is pushed on the stack and sets the indicators.
23 CDDR	The CDDR of CE is pushed on the stack and sets the indicators.
24 CDAR	The CDAR of CE is pushed on the stack and sets the indicators.
25 CAAR	The CAAR of CE is pushed on the stack and sets the indicators.
26 SCDR	Get the CDR of CE and store it in E and the indicators. Useful for CDRing down lists. CE must be a list cell.
27 SCDDR	Get the CDDR of CE and store it in E and the indicators. Useful for CDDRing down property lists.
30 Trunc	Performs the equivalent of the TRUNC function as described in the Spice Lisp Manual. After obtaining CE, take one value off the top of the stack. Call this N. There are now three cases: • If CE is Fixnum 1, Trunc pushes three items: a fixnum or bignum representing the integer part of N (rounded toward 0), then either 0 if N was already an integer or the fractional part of N represented as a flonum with the same type as N, then Misc-Values-Marker 2 to mimic a multiple return of two values.

- If CE and N are both fixnums or bignums and CE is not 1, divide N by CE. Push three items: the integer quotient (a fixnum or bignum), the integer remainder, and Misc-Values-Marker 2.
- If either CE or N is a flonum, push a fixnum or bignum quotient (the true quotient rounded toward 0), then a flonum remainder equal to (- N (* CE QUOTIENT)), then Misc-Values-Marker 2.

In all cases the first thing pushed (the quotient) sets the indicators.

31 +	CE is added to the value popped off the stack. The result is pushed back onto the stack and sets the indicators. The addition is generic: two integers (fixnum or bignum) produce an integer; two ratios or an integer and a ratio produce a normalized ratio; if either operand is a flonum, the result is a flonum of that type, long or short; if one of the operands is a long flonum and the other is a short flonum, the result is a long flonum.
32 -	Analogous, but CE is subtracted from TOS. Generic, as above.
33 *	Analogous, CE is multiplied by TOS. Generic, as above.
34 /	The TOS is divided by CE; the quotient goes back to TOS. This is generic, as above, except when the operands are both integers. In that case, an integer is returned only if the division is exact (that is, if the remainder is 0). If the division is not exact, the quotient is returned as a normalized ratio.
35 Bit-And	Bitwise boolean AND of CE and top of stack. The result goes onto the stack and sets the indicators. The operands must be fixnums or bignums.
36 Bit-Xor	Bitwise XOR.
37 Bit-Or	Bitwise OR.
40 Eql	CE is compared to the value popped off the stack. If these arguments are EQ or if they are both numbers of identical type and value, T sets the indicators; if not, () sets the indicators. Nothing is pushed back onto the stack.
41 =	CE is compared arithmetically to the value popped off the stack. If they are equal, T sets the indicators; if not, () sets the indicators. Nothing is pushed back onto the stack. This works for mixed number-types: if an integer is compared with a flonum, the integer is floated first; if a short flonum is compared with a long flonum, the short one is first extended. Flonums must be exactly identical (after conversion) for a non-null comparison.
42 >	Analogous, but non-null if TOS > CE.
43 <	Analogous, but non-null if TOS < CE.
44 EQ	CE is compared to the value popped off the stack. If these objects are identical 32-bit Lisp objects, T sets the indicators; if not, () sets the indicators. Note: equal fixnums and equal

short flonums are EQ, but this does not hold for bignums or long flonums. So when dealing with numbers use = or EQL unless you are sure you know what you are doing.

45 1+	Add 1 to CE, store result back into E. CE can be any kind of number.
46 1-	Subtract 1 from CE, store result back into E. CE can be any kind of number.
47 Bind-Null	CE must be a symbol. This is rebound and set to (). The NULL indicator is set.
50 Bind-T	CE must be a symbol. This is rebound and set to T, which also sets the indicators.
51 Bind-Pop	CE must be a symbol. This is rebound and is set to a value popped off the stack. This value also sets the indicators.
52 Set-Null	Store () in E.
53 Set-0	Store fixnum 0 in E.
54 Set-T	Store T in E.
55 - 57 Unused.	
60 NPop	CE is a fixnum N. If N is non-negative, N items are popped off the stack. If N is negative, () is pushed onto the stack |N| times. The indicators are unchanged.
61 Unbind	CE is a fixnum indicating how many bindings are to be popped off the binding stack and restored to their previous values. Used in exiting open-coded PROGs and LAMBDAs. The indicators are unchanged by this instruction.
62 Set-Lpush	Pop TOS, cons it onto CE (in the space indicated by the value of the symbol ALLOCATION-SPACE), store result back into E. The new CE sets the indicators.
63 Set-Lpop	CAR of CE is pushed onto the stack and sets the indicators; CDR of CE is stored back into E.
64 List	CE is a fixnum N. Beginning with a list of (), N items are popped off the stack and CONSed onto this list, so that the last item popped ends up as the CAR of the list. The consing is done in the space specified by the value of ALLOCATION-SPACE. The resulting list is pushed on the stack and sets the indicators.
65 List*	CE is a fixnum N. One item is popped off the stack, to begin the list L. Then N other items are popped and CONSed onto the front of L in succession, so that the last item popped becomes the CAR of L. The consing is done in the space specified by the value of ALLOCATION-SPACE. The resulting list is pushed onto the stack and sets the indicators.
66 Spread	CE is a list. Its elements are pushed onto the stack in left-to-right order. The last item pushed sets the indicators.

67 Misc — This is used for calling a large number of microcoded functions. The next byte in the instruction stream is fetched, and this is used to indicate which of 256 Misc functions is to be called. This operation will in general pop some arguments off the stack, compute a single result, then place this result in the location indicated by the effective address E, computed as usual from the A field of the first byte. Note that if one or more offset bytes are needed for the effective address computation, these bytes are fetched *after* the byte telling which instruction is to be called. The Misc codes are defined in a later section of this document. (Initially, many fewer than 256 Misc functions are defined.)

70 Branch — Unconditional branch relative to the current byte-PC (which has been incremented to point past the current instruction). The next byte or two bytes is fetched. This, treated as a signed integer, is added to the PC. The indicators are unchanged. For all of the branch instructions, the bits of the A field are interpreted as follows:

Bit 0 = 0	Fetch one byte for branches of -128 to +127 bytes.
Bit 0 = 1	Fetch two bytes for longer branches. The low-order byte comes in first.
Bit 1 = 0	Do not pop stack.
Bit 1 = 1	Pop stack if the (conditional) branch is *not* taken.

71 Branch-If-Arg-Supplied
This is a special conditional branch that is used by the machinery that computes default values for optional function arguments that were not supplied by the caller. The next byte is read from the instruction stream and is taken as an offset (range 0 - 255) into the args-and-locals area of the stack frame. If the stack frame entry in question contains Misc-Unsupplied-Arg, do not branch; otherwise, take the branch. The branch is executed normally, using the A-field of the instruction to control the usual branch options. The branch offset byte(s) will follow the argument offset byte in the instruction stream.

72 Branch-Null — Branch if the Null indicator is on. Does not alter indicators (nor do any of the other branches).

73 Branch-Not-Null
Branch if the Null indicator is not on.

74 Branch-Atom — Branch if the ATOM indicator is on.

75 Branch-Not-Atom
Branch if the ATOM indicator is not on.

76 Branch-Zero — Branch if the ZERO indicator is on.

77 Branch-Not-Zero
Branch if the ZERO indicator is not on.

5.8. MISC Instructions

The following instructions are members of the MISC group. Each of these expects a fixed number of arguments to have been pushed on the stack in the order indicated (leftmost arg pushed first). These arguments are popped and a single return value is generated. This sets the indicators and goes to the E location of the Misc operation. The numbers in the left margin are the 8-bit octal codes corresponding to each instruction.

The user can access these functions directly from compiled Lisp code and can indirectly achieve the same effect from interpreted code. See section 5.9 for details.

000 Cons (X Y) Conses up a list cell with X as CAR and Y as CDR. The allocation space is controlled by the value of the symbol ALLOCATION-SPACE.

001 Alloc-Symbol (N)
Allocates one symbol and returns a pointer to it. The allocation space is controlled by the value of ALLOCATION-SPACE. String N becomes the symbol's PNAME. The value is initially Misc-Trap, the definition is Misc-Trap, the plist, package, and hash are initially (). The symbol is not interned by this operation -- that is done in macrocode.

002 Alloc-B-Vector (N I)
Allocates a B-Vector of N entries and returns a pointer to it. Allocation space is controlled by the value of ALLOCATION-SPACE. I is the initial value with which the vector is filled.

003 Alloc-U-Vector (N A)
Allocate a local U-Vector with access-code A and a length of N items, and return a pointer to it. Allocation space is controlled by the value of ALLOCATION-SPACE. All entries are initialized to 0.

004 Alloc-Remote-Vector (N A P)
Allocate a remote U-Vector with N entries and access-code A, returning a pointer to it. P is the pointer to the data area in system-table space. Allocation space is controlled by the value of ALLOCATION-SPACE.

005 Alloc-String (N)
Allocate a string of length N, initialized to all 0's, and return a pointer to it. Allocation space is controlled by the value of ALLOCATION-SPACE.

006 Alloc-Function (N)
Allocate a function object (like a B-Vector) of length N, not counting the 1-word header. Allocation space is controlled by the value of ALLOCATION-SPACE. Initialized to 0 (Misc-Trap codes).

007 Alloc-Array (N)
Allocate an array-header for an array of N dimensions. Allocation space is controlled by

the value of ALLOCATION-SPACE. The header is initialized to 0 (Misc-Trap codes), and must be filled with the appropriate header info by other code. Returns a pointer to the array header.

010 Alloc-Xnum (N X)
Allocate an xnum N bytes in length, with sub-type code X. N and X must be fixnums. The allocation space is controlled by the value of ALLOCATION-SPACE. All entries of the XNUM vector are initialized to 0.

011 Alloc-Ynum (N X)
Allocate a Ynum N lisp-objects in length, with sub-type code X. N and X must be fixnums. The allocation space is controlled by the value of ALLOCATION-SPACE. All entries of the YNUM vector are initialized to Misc-Trap codes.

012 Misc-Subtype (X)
X must be of type MISC. Returns the subtype field (bits 24-27) of X right-justified in a fixnum.

013 Type (X) Returns the 4-bit type-code of X as a fixnum.

014 Make-Immediate-Type (OBJ TYPE)
OBJ can be any lisp object, TYPE is a fixnum in the range 0 - 2, which correspond to the type-codes of immediate objects. Returns an object whose type-code bits are TYPE, but whose other bits are those of OBJ. Used for converting fixnums to misc-type objects, converting pointer-type objects to fixnums so that the pointer can be examined, etc.

015 Get-Vector-Subtype (V)
Returns the 4-bit subtype field of a vector-like object V (B-vector, U-Vector, Array, Xnum, String, Function). Returned as a fixnum.

016 Set-Vector-Subtype (V X)
Stores the low order 4 bits of fixnum X as the subtype code of vector-like thing V. Returns V.

017 Get-Vector-Length (V)
V is any vector-like thing. Returns the length of this vector, which is one greater than the largest legal index, as a fixnum.

020 Get-Value (S) Gets the contents of the value cell of the symbol S. Signals an UNBOUND error if the value is Misc-Trap.

021 Set-Value (S V)
Set the value cell of symbol S to V. If the cell contains an EVC-Forward pointer, this is followed. Returns V.

022 Get-Definition (S)
Returns the contents of the functional definition cell of symbol S. Signals an UNDEFINED error if the cell contains Misc-Trap.

023 Set-Definition (S D)	Puts D into the functional definition cell of symbol S. Returns D.
024 Get-Plist (S)	Returns the property list of symbol S.
025 Set-Plist (S P)	Sets the property list of symbol S to P. P should be () or a List object. Returns P.
026 Get-Pname (S)	Returns the pname of symbol S.
027 Set-Pname (S P)	Sets the pname of symbol S to P. P should be a string. Returns P.
030 Get-Package (S)	Gets the contents of the package cell of symbol S.
031 Set-Package (S P)	Sets the package of symbol S to P. Returns P.
032 Get-Hash (S)	Gets the contents of the hash cell of the symbol S.
033 Set-Hash (S H)	Set the hash cell of symbol S to H. Returns H.
034 Boundp (S)	S must be a symbol. Boundp returns () if the value cell of the symbol constains Misc-Trap, T otherwise.
035 Fboundp (S)	S must be a symbol. Fboundp returns () if the definition cell of the symbol constains Misc-Trap, T otherwise.
036 Rplaca (L X)	Replaces car of L with X, returning the modified L.
037 Rplacd (L X)	Replaces cdr of L with X, returning the modified L.
040 Unused.	
041 S-Float (X)	Turns any number X into a short flonum.
042 L-Float (X)	Turns any number X into a long flonum.
043 Negate (X)	For any number X, return the negative.
044 Lsh (N B)	Both args are fixnums. Returns a fixnum that is N shifted left by B bits, with 0's shifted in on the right. If B is negative, N is shifted to the right with 0's coming in on the left.
045 Get-Vector-Access-Type (V)	V must be a U-Vector. Returns the access-type code (bits 28-31 of the second header

word) right-justified in a fixnum.

046 Logldb (P S N)
All args are fixnums. P and S specify a "byte" or bit-field of any length within N. This is extracted and is returned right-justified in a fixnum. S is the length of the field in bits; P is the number of bits from the right of N to the beginning of the specified field. P = 0 means that the field starts at bit 0 of N, and so on. If the specified field is not entirely within the 28 bits of N, it is an error.

047 Logdpb (V P S N)
All args are fixnums. Returns a number equal to N, but with the field specified by P and S replaced by the S low-order bits of V. An error if the field does not fit into the 28 bits of N.

050 Abs (N) N is any kind of number. Returns the absolute value of N.

051 Subspace (X) X is any lisp object. Returns the 2-bit allocation space code as a fixnum. Returns () if the object is immediate.

052 Close-Over (L)
L is a list of symbols. Creates and returns a closure-list for these symbols in the current environment. See section 6.8 for details.

053 Activate-Closure (C)
C must be a closure list, as returned by the Close-Over operation. Activate-Closure restores the environment in which the closure list was created for the symbols closed over. See section 6.8 for details. Returns C unchanged.

054 Typed-V-Access (A V I)
A and I are fixnums, V points to a U-Vector or Xnum. This returns entry I of the V as a fixnum, but uses the low-order three bits of A as the access-type code instead of whatever code is stored in the vector itself. The high-order bit of the access-type code, controlling whether the vector is remote, is always taken from the vector itself. This is illegal if V is a string. Bounds checking should be done by the caller, though a reference to a field that is grossly out of bounds for this vector may be caught.

055 Typed-V-Store (A V I X)
Like a V-Store, but stores X in entry I of V using A as the low-order 3 bits of the access-type code, as above. Returns X. Illegal for strings.

056 Unused.

057 Freeze () Freezes all read-only spaces by moving the FREEZE pointers up to meet the FREE-STORAGE pointers. Returns ().

060 New-Pure-Page (X)
X can be an item of any non-immediate data type. The type of X is examined, and the current read-only page for that type of storage is closed. This is done by moving the FREE-STORAGE pointer for that space up to the next page boundary if it is not on a page

bondary already, and moving the FREEZE pointer to the same place. Returns X.

061 Shrink-Vector (V N)

V is any B-Vector, U-Vector, String, Function object, or Array header. N is the new number of entries, a fixnum, which must be less than or equal to the current number of entries. The length field and the number-of-entries field of the vector are altered to reflect this new shorter length. If the object contains boxed entries, those beyond the new end of the vector are set to 0 (Misc-Trap codes). Returns V, the vector which has been shortened.

062 Call-Break (F)

Just like the Call operation, but starts the new frame with a break-type Misc-Frame-Header word (bit 23 = 1) rather than the usual kind of header. This means that when the called function ultimately returns, no return value is left on the stack. Gets F, the function to be called, from the stack; returns (), though this will normally be called with destination IGNORE.

063 Values-To-N (V)

V must be a Misc-Values-Marker. Returns the number of values indicated in the low 24 bits of V as a fixnum.

064 N-To-Values (N)

N is a fixnum. Returns a Misc-Values-Marker with the same low-order 24 bits as N.

065 Arg-In-Frame (N F)

N is a fixnum, F is a control stack pointer as returned by the CURRENT-STACK-FRAME and CURRENT-OPEN-FRAME operators. Returns the item in slot N of the args-and-locals area of stack frame F.

066 Current-Stack-Frame ()

Returns a control-stack pointer to the start of the currently active stack frame. This will be of type Misc-Control-Stack-Pointer.

067 Set-Stack-Frame (P)

P must be a control stack pointer. This becomes the current active frame pointer. Returns ().

070 Current-Open-Frame ()

Returns a control-stack pointer to the start of the currently open stack frame. This will be of type Misc-Control-Stack-Pointer.

071 Set-Open-Frame (P)

P must be a control stack pointer. This becomes the current open frame pointer. Returns ().

072 Current-Stack-Pointer ()

Returns the Misc-Control-Stack-Pointer that points to the current top of the stack (before the result of this operation is pushed). Note: by definition, this points to the first unused word of the stack, not to the last thing pushed. The stack manipulation instructions make it appear as if the stack is all in contiguous virtual memory, despite the fact that in some implementations a TOS register or some other form of hardware buffer will be holding part of the stack.

073 Current-Binding-Pointer ()
Returns a Misc-Binding-Stack-Ptr that points to the first word above the current top of the binding stack.

074 Read-Control-Stack (F)
F must be a control stack pointer. Returns the lisp object that resides at this location. If the addressed object is totally outside the current stack, this is an error.

075 Write-Control-Stack (F V)
F is a stack pointer, V is any Lisp object. Writes V into the location addressed. Returns V. If the addressed cell is totally outside the current stack, this is an error. Obviously, this should only be used by carefully written and debugged system code, since you can destroy the process using this operation.

076 Read-Binding-Stack (B)
B must be a binding stack pointer. Reads and returns the lisp object at this location. An error if the location specified is outside the current binding stack.

077 Write-Binding-Stack (B V)
B must be a binding stack pointer. Writes V into the specified location. Returns V. An error if the location specified is outside the current binding stack.

100 Ldb (P S N) All args are fixnums or bignums; P and S are non-negative. P and S specify a "byte" or bit-field of any length within N. This is extracted and is returned right-justified as a positive integer. S is the length of the field in bits; P is the number of bits from the right of N to the beginning of the specified field. P = 0 means that the field starts at bit 0 of N, and so on. N is considered to extend infinitely to the left; therefore, if the specified field extends to the left of the internal representation of N, the remaining bits are filled with copies of the sign bit of N.

101 Mask-Field (P S N)
Like LDB, except that the extracted field is returned in the same position it occupies in N, not moved to the right. The result is a positive fixnum or bignum with 0 in all positions except that specied by the S-P field.

102 Dpb (V P S N)
All args are fixnums or bignums; P and S are non-negative. Returns a number equal to N, and with the same sign as N, but with the field specified by P and S replaced by the S low-order bits of V. The result may be extended to the left of the field to make the sign come out right.

103 Deposit-Field (V P S N)
Like DPB, except that the bits to be put in N are extracted from the corresponding field of V, not from the rightmost S bits of V.

104 Ash (N C) N and C are fixnums or bignums. Shift N left C places, shifting in zeros on the right. If C is negative, shift N right -C places, preserving the sign of N.

105 Haulong (N) N is a fixnum or bignum. Returns the number of significant bits of N.

106 V-Access (V I)
V is any vector or vector-like object (B-Vector, U-Vector, String, Xnum, Array, or Function Object). I is a Fixnum. Returns entry I of V. If the vector is a U-Vector or Xnum, the item returned is a fixnum; if a string, the item returned is a Char Object with bits and font code of 0.

107 V-Store (V I X)
V is any vector or vector-like object. I is a fixnum. X is the value to be stored into slot I of vector V. For U-vectors and Xnums, X must be a fixnum, and is truncated to fit the vector item size; for strings, X must be a Misc-Character code, and only the 8-bit code portion is saved. X is returned.

110 - 117 Unused. Reserved for I/O operators.

120 Force-Values ()
If the top of the stack is a Misc-Values marker, do nothing; if not, push Misc-Values 1. Returns (), but normally this will be called with destination IGNORE.

121 Flush-Values ()
If the top of the stack is a Misc-Values marker, remove this marker; if not, do nothing. Returns (), but normally this will be called with destination IGNORE.

122 Mark-Catch-Frame ()
Set bit 20 of the header word of the current *open* frame, marking this as a catch-tag frame. Returns ().

Misc-codes still to be defined: control of flipping and scavenging, interface to IPC and kernel...

5.9. Sub-Primitives

It is intended that no Spice Lisp code will be written directly in the macro-instruction set; everything not microcoded will be written in Spice Lisp itself. Since the macro-compiler for Spice Lisp will run in Maclisp as well, we do not even need to write macrocode for bootstrapping. This requires that some system-level operations not available to normal users of Spice Lisp must be available within the Spice Lisp language to the system implementors. Such system-level Lisp functions are called sub-primitives, and are identifiable by the prefix %SP-. Since we are not attempting to be compatible with the Lisp Machine at the sub-primitive level, we have deliberately chosen a different prefix to avoid confusion.

All of the Miscellaneous operations are accessible as sub-primitives. When the compiler sees something like

```
        (%SP-Type <arg>)
```

it generates something like

```
        Push <arg>                  ;Push the single argument.
        Misc Type                   ;Call the Type Miscop.
                                    ;Return normally left on stack.
```

Of course, the compiler is free to use these misc operations in other situations as well. In a few cases, such as RPLACD, the misc operation will correspond directly to a user-visible function. In such a case, both a RPLACD and a %SP-RPLACD form will exist, and the compiler will generate the same code for both. In general, sub-primitive functions will be undefined if they are encountered in non-compiled code, though we will create a file of dummy definitions for the purpose of debugging uncompiled system code. (This file, at least, must be compiled.)

A number of additional sub-primitives exist that do not expand directly into misc-ops. See the section on Sub-Primitives in the Spice Lisp User's Manual for details.

6. Internal Control Conventions

6.1. Control Registers

The following 32-bit registers are kept within the processor and are accessed by microcode. All of these are saved and restored across process breaks and kernel calls. Those registers that are "inside the barrier" and whose contents must be transported by the flipper are marked with an asterisk (*).

Flags	28 single-bit state indicators, used for assorted purposes by microcode. The high 4 bits contain a fixnum type-code so that this word can be pushed onto the stack if necessary. (Actually, this code might not be present internally, but only when the flags word is written into boxed storage.)
TOS (*)	The top (most recently pushed) entry of the control stack, unless the stack is empty. Kept in a register for efficiency.
Control-Stack-Ptr	The stack pointer for the main or control stack. Points to the first unused word of control-stack space. That is, the upward-growing stack uses a write-increment/decrement-read discipline. The high 8 bits of this word contain a Misc-Control-Stack-Ptr type-code. (Again, this code may not be present internally, but only when the register is written into memory.)
Binding-Stack-Ptr	The stack pointer for the special variable binding stack. Same discipline as for the Control-Stack-Ptr. The type code is Misc-Binding-Stack-Ptr.
Active-Frame	Points to the first word of the control stack frame for the function currently executing. Type Misc-Control-Stack-Ptr. Note: the virtual address of the start of the Args-and-Locals area of the active frame is this pointer plus a constant (see section 6.3).
Open-Frame	Points to the first word of the control stack frame being built (args being evaluated) but not yet entered. Type Misc-Control-Stack-Ptr.
Active-Function (*)	Points to the boxed function object for the currently executing function. Note: the virtual address of the start of the Symbols-and-Constants area of the current function is this pointer plus a constant. See section 5.2 for the value of this offset.
Active-Code (*)	Points to the unboxed code vector for the currently-executing function. Note: only the flipper can move this object, since to get into this register it must already be in newspace.
PC	Conceptually, this is a fixnum that indicates which byte of the code-vector contains the next instruction to be executed. This is the form in which the PC is stored externally. The internal format is machine-specific. On the PERQ, the PC is stored internally as an offset (ending in three zeros) to the start of the PERQ quad-word containing the next byte code and a 3-bit entry in the hardware BPC

indicating which byte of this quadword is to be read or executed next. The quadword in question is cached in the instruction buffer of the PERQ. Note that the Current Code object can only be moved by the flipper, and that the PC, BPC, and instruction buffer contents do not need to be altered when such a movement occurs. The operation of computing a virtual address and reloading the instruction buffer must be atomic with respect to flipping, but this is easy to achieve. All of this depends on the fact that the old and new versions of the code vector have the same quad-word alignment; this is assured since all objects in the PERQ implementation are quad-word aligned.

6.2. Binding Stack Format

Each entry consists of two boxed (32-bit) words. Pushed first is a pointer to the symbol being bound. Pushed second is the symbol's old value (any boxed item) that is to be restored when the binding is popped.

6.3. Control Stack Frame Format

Each function call creates a new frame on the control stack. If the function is a compiled Function Object, the stack frame is as follows. The entry marked 0 is the word pointed to by the Active-Frame register if this frame is current.

```
0   Header word.  Type Misc-Frame-Header.
1   Function object or EXPR for this call.
2   Closure List (or () if not a closure).
3   Pointer to previous active frame.  Type Misc-Control-Stack-Ptr.
4   Pointer to previous open frame.  Type Misc-Control-Stack-Ptr.
5   Pointer to previous binding stack.  Type Misc-Binding-Stack-Ptr.
6   Saved PC of caller.  A fixnum.
7   Args-and-locals block starts here.  This is entry 0.
...
N   Misc-Frame-Barrier.  Push after the args and locals.
```

6.4. Call Instruction

This macro-instruction opens a call block on the stack, but does not actually begin the call. Instead, the arguments are evaluated and pushed on the stack, and the call is actually started by the PUSH-LAST instruction when the last argument is in place. Note that in evaluating the arguments for one function, other functions may have to be called, so a chain of several open call blocks may be present at one time.

The CALL instruction receives the function to be called as CE. It then checks the type of CE and proceeds as follows, according to the type:

If CE is a symbol, fetch the contents of the symbol's definition cell. If this is Misc-Trap or another symbol, signal an error. Else, go on from here as if this were the original CE. (We cannot allow chains of symbols defined as other symbols, since this could lead to a very nasty kind of infinite loop that would be expensive to check for.)

If CE is a function object, perform the following steps:

1. Note the current value of Control-Stack-Ptr.
2. Push Misc-Frame-Header on control stack.
3. Push CE (the function being called).
4. Push () as the closure list.
5. Push Active-Frame.
6. Push Open-Frame.
7. Push Binding-Stack-Ptr.
8. Push Fixnum -1 (this will later be filled with caller's PC).
9. Open-Frame <= = Stack frame pointer saved in step 1.

The new open frame is now ready to have arguments pushed by code in the calling function, terminated by a PUSH-LAST to start the call.

If CE is a list, and its CAR is of type Misc-Closure-Marker, then this is a closure. Its CADR is the function or Lambda to be called, and its CDDR is the closure list. Proceed with the call as if the CADR were the original CE, but push the CDDR into entry 2 of the stack frame instead of ().

If CE is a list whose CAR is not of type Misc-Closure-Marker, it is probably a LAMBDA expression. The call proceeds exactly as specified above, with the list stored in the function slot of the new frame. The arguments will be pushed normally, then %SP-INTERNAL-APPLY will be called when PUSH-LAST is executed. (See below.) %SP-INTERNAL-APPLY will verify that this is in fact a legal LAMBDA or something else that it knows how to handle.

If CE is anything else, an ILLEGAL-FUNCTION error is signalled.

6.5. The Push-Last Instruction

This pushes CE (unless A = 0, in which case the stack is unchanged) and starts the function responsible for the current open frame. Note that CALL-0 opens the frame, then jumps in here immediately, since there are no args to set up.

If Open-Frame is null, signal an error. If there is a frame, and it is for a compiled function (entry 1 of the open frame is of type Function), proceed as follows:

1. Insert the current PC (points to the NEXT instruction of the caller's code vector) in the PC slot of the open frame.
2. Active-Function <= = Called function (from slot 1 of open frame).
3. Active-Code <= = Code vector for new active function.
4. Note number of args pushed by caller.
5. If number of args < minimum, signal an error.
6. If number of args > maximum and a &REST arg is present, pop excess args into a list, push this list back on stack as the &REST arg, then go to step 9.
7. If number of args > maximum and no &REST arg, signal an error.
8. If number of args is between min and max (inclusive) push a MISC-UNSUPPLIED-ARG for each remaining optional and () for the &REST arg, if any.
9. Push () for as many locals as the function requires.
10. Push Misc-Frame-Barrier to finish frame.
11. Active-Frame <= = Open-Frame
12. Open-Frame <= = ()
13. Set up PC = 0 (Note: this points to entry 0 in the code vector, which is always the address of the first instruction. Internally on the Perq, we must actually set up PC, BPC, and fill the instruction buffer.)
14. If slot 2 (the closure list) of the current frame is non-null, perform an ACTIVATE-CLOSURE operation on this list.
15. Return to macro-code execution loop to run new function.

If the object in entry 1 of the open frame is a lambda expression rather than a function object, we must call the %SP-INTERNAL-APPLY function to interpret this expression with the given arguments. To achieve this we proceed as follows:

1. Note the number of args pushed in the current open frame (call this N) and the frame pointer for this frame (call it F). Also remember the lambda-expression in this frame (call it L).

2. Perform steps 1 and 10 - 14 of the sequence specified above for a normal PUSH-LAST.

3. Perform the equivalent of a CALL-MAYBE-MULTIPLE instruction with the symbol %SP-INTERNAL-APPLY as CE. (This symbol is in a fixed location known to the microcode. See section 2.8.)

4. Push L, N, and F in that order as the three arguments to %SP-INTERNAL-APPLY.

5. Perform the equivalent of a PUSH-LAST with A = 0 to start the call.

%SP-INTERNAL-APPLY, a compiled function of three arguments, now evaluates the lambda expression L, obtaining the arguments from the frame pointed to by F. These arguments are obtained using the ARG-IN-FRAME Misc-op. Just prior to returning, %SP-INTERNAL-APPLY sets the ACTIVE-FRAME register to F, so that it returns from frame F.

6.6. Return Instruction

Returns from the current function, popping the stack frame and then pushing some number of returned values, as determined by the nature of CE and the bits set in the frame header word.

If CE is any type of object except a Misc-Values-Marker, the function is trying to return a single value, namely CE. If the current frame begins with a break-type frame header, CE is discarded and the saved indicator values are restored. If the current frame begins with an escape-to-macro type header, CE is placed in the location indicated by the codes stored in the header word. If the frame begins with a multiple-value frame header, the single value is returned and the Misc-Values-Marker with 1 in the low 24 bits is pushed on top of it, indicating that 1 value is being returned.

The steps are as follows:

1. Pop binding stack back to value saved in slot 5 of the active control frame. For each symbol/value pair popped off the binding stack, restore that value for the symbol.

2. Temp <== Previous active frame from slot 3 of current frame.

3. Open-Frame <== Saved value in current frame.

4. PC <== Saved value in current frame. Note that on the PERQ this requires setting up the internal PC, the BPC, and the instruction buffer.

5. Active-Function <== Saved value from previous frame. A pointer to this frame is in Temp.

6. Active-Code <== Code Vector obtained from entry in restored Active-Function object.

7. Pop current frame off stack:

   Control-Stack-Pointer <== Active-Frame.
   Active-Frame <== Temp.
   Pop top of stack into TOS register.
   Since the active frame is inside the barrier, make sure the new top frame has been scavenged, or do it now.

8. If frame exited was normal (began with Misc-Frame-Header), push the return value onto the stack and use it to set the indicators.

9. If frame exited was of break type, restore the indicators from the values saved in the header word, but push nothing on the stack.

10. If frame exited was of escape-to-macro type, place the return value wherever the saved A field and offset indicate that it should go. The return value also sets the indicators.

11. If frame exited was of multiple-value type, push the return value on the stack and use it to set the indicators. Then push Misc-Values-Marker 1 to indicate that only one value was returned.

12. Resume execution of function popped to.

If RETURN is called with CE of type Misc-Values-Marker, the low 24 bits of CE indicate how many values are to be returned; call this number N. The N things to be returned have already been pushed as the top N items on the stack. In this case the return proceeds as follows:

1. Note the value of the current stack pointer (after CE is popped off if it came from the stack) as OLDSP.

2. Perform steps 1 - 7 of the RETURN protocol described above. Do not release the space occupied by the old stack frame back to the system, since we will need to get at the return values that are still in the old frame. (This space is not currently released anyway.)

3. Examine the header word of the frame being returned from to determine the type of the call. Proceed as follows:

   a. For an ordinary call, if N = 0 push () as the single return value. Else, pick up the first return value from location (OLDSP) - N and push that as the return value. In either case, the thing returned sets the indicators.

   b. For escape-to-macro call, proceed as for an ordinary call but put the return value wherever the header word indicates, not necessarily on the stack.

   c. For break-type call, do just what an ordinary RETURN would do: restore the saved indicators and push no return value.

   d. For multiple-value call, do a block transfer loop pushing the N words starting at (OLDSP) - N onto the stack as return values. Then push the original CE, which is Misc-Values-Marker N. The first returned value sets the indicators, or () if N is 0.

4. Resume execution of the caller.

6.7. Handling Optional Arguments

Spice Lisp supports &OPTIONAL, &REST, and &AUX arguments like those in the Lisp Machine. We do not plan to support the &QUOTE mechanism; all arguments to a function are evaluated. (The only exception to this is in functions of the FEXPR type, in which the entire argument list is passed unevaluated to the function as its single argument.) Some arguments and &AUX variables are local (lexically bound on the stack) while others are special (shallow-bound in the value cell of the symbol).

In the Lisp Machine this variability is handled by a variety of rather complex mechanisms which we cannot afford to duplicate in our more limited microstore. Instead, we handle the variation in a uniform way, primarily within the macrocode of the function. As has already been described, the function object contains information indicating the maximum and minimum number of arguments and whether there is a &REST argument. At the time a call is started, all of the arguments (whether special or not) are present on the stack, with MISC-UNSUPPLIED-ARG codes in the place of any optional arguments that were not supplied by the caller. Space has also been allocated for any non-argument locals, and these have been initialized to (). At this point the function's macrocode is entered at PC = 0.

Note that all of the arguments passed by the caller are executed in the caller's environment, but that in evaluating default value expressions for unsupplied arguments and initialization expressions for &AUX variables, the assumption is that evaluation and binding proceeds left to right: each initialization expression can assume that the variables to its left have already been set up.

If an &optional argument is a 3-element list (FOO INIT FOOP), the FOOP value gets set to () if the caller supplied no argument, T if the argument was supplied. In the description below, the third element will be called a "foop" parameter.

The macrocode proceeds through the following steps:

1. Bind any special variables associated with required arguments, to their new values. The arguments are left on the stack as passed, but future accesses and SETQs are to the value cell of the symbol.

2. For each optional argument, in order from left to right, the following code is generated:

```
          If there is a FOOP, set it to T if it is local,
            bind it to T if it is special.
          Branch-If-Arg-Supplied  Arg-number  Dest
          If there is a FOOP, set it to ().
          <Code to initialize argument on stack>
  Dest:   <Code to bind argument to value on stack if arg is special>
```

3. Code to bind the &REST argument if it exists and is special.

4. Code to initialize &AUX variables, and to bind those that are special. These are initialized and bound in left-to-right order, so that the initialization code for each variable can use the values of variables to its left.

5. Code for the function itself.

6.8. Closures

When a closure of a function is created, the binding environment in which the closure was created is saved; when the closure is called, any references to special variables from within the closed function will see the versions of the variables that existed at the time the closure was created, rather than the current execution environment. This is true even if the closing environment has been exited, which under normal conditions would have popped and destroyed this environment.

In Spice Lisp we use a closure scheme invented by the MIT Lisp Machine group. It saves the environment only for a set of variables which the user specifies, and not the entire environment present at the time of the closure. At the time of the creation of the closure, an External Value Cell (EVC) is created in list space for each symbol closed over. The symbol's current value is placed in the EVC, and an EVC-Forward pointer is placed in the value slot of the symbol. Any attempt to reference or alter the value of the symbol will now be forwarded by this pointer and will reference or alter the value in the EVC instead. The EVC (but not the forwarding pointer) survives any popping of the current binding environment. When the closure is called, the EVCs for all the closed variables can be set up again by the creation of new EVC-Forward pointers. The existence of closures thus costs nothing if this feature is not used, and costs relatively little even when closures are used. The two key operations in dealing with closures are Close-Over and Activate-Closure.

Close-Over is a miscellaneous operation that takes one argument, a list of symbols that are to be closed over in the current environment. It returns a closure-list, which is a list with the following format:

```
((value1 . symbol1) (value2 . symbol2) ... (valueN . symbolN))
```

This list is built by performing the following steps for each symbol S in the argument list:

1. Get the contents of the value cell of S. Call this V. Do not follow EVC-Forward pointers in this

fetch.

2. If V is of type EVC-Forward, get a (non-forwarding) pointer to the list cell that this points to. CONS that onto the closure list to be returned. (If two different closures are created of the same binding of a symbol, we share the EVC rather than letting the EVC-Forward pointers chain.)

3. Else, CONS up a new list cell in dynamic space, putting V in the CAR and S in the CDR. Cons this cell onto the list to be returned. Create an EVC-Forward pointer to the new cell and put this in the value cell of S.

Once the closure list is created, it is turned into a function closure by CONSing on the function and a Misc-Closure-Marker. This is done in macro-code.

The Activate-Closure function takes a closure list as its single argument and restores the environment that this closure list represents. It does this by CDR-ing down the closure list, creating an EVC-Forward pointer to each element in the list and rebinding the symbol in this cell to the EVC-Forward pointer. The previous values of these symbols are saved on the binding stack, as in all rebindings. Activate-Closure is called internally by the Push-Last operation when the function being started is a closure.

6.9. The Escape to Macro Convention

Spice Lisp will be implemented on a variety of machines, not all of which will have enough microstore space for all of our needs. Some of the macro-instructions that we have defined are relatively simple for most cases, but can be arbitrarily complex for some cases. For example, the arithmetic instructions can handle fixnums and (maybe) flonums in microcode, but (on the Perq, at least) we would like to escape to a macrocode routine if we discover that we have to deal in bignums.

Such cases could be handled by a full-scale microcode-to-macrocode subroutine call, which upon a return comes back to the designated return address in the microcode and restores any micro-state that may have been clobbered. This may ultimately be needed if we ever implement a micro-compiler for lisp, but for now we can get by with a simpler scheme. If the microcode for any macro-instruction decides that it has a case too difficult to handle, it can call a macrocoded function that does whatever the original macro-instruction was supposed to do. It does this by opening an escape-type frame on the control stack, pushing an appropriate set of arguments, and then starting the call as though a push-last had been done in macrocode. When the macrocoded escape function returns, the returned value goes wherever the original macro-instruction was supposed to place its result, and the original instruction stream continues on as if the macrocode instruction had exited normally without an escape.

Macro and misc instructions can place their return values in any of several destinations: push on the stack, throw away but set the condition codes, store in the current stack frame, or set the value of a symbol. The escape call must set up the frame header word to indicate which of these locations is to get the value returned by the macro-coded escape function. The appropriate A value is stored in bits 16-17. The offset, if any, is stored in bits 0-15. A long (two byte) offset is always used here, since we don't have room to alternate between one and three bytes. Given this information in the frame header, RETURN will do the right thing to make it appear that the original macro-instruction had exited normally.

The macro-coded escape functions all live in read-only function space. A table of pointers to these functions is stored in a fixed location in physical memory, and the address of the start of this table is known to the microcode. This means that microcode routines can select the desired function by means of a table index, and it is not necessary to assemble the addresses of all these functions into the microcode.

The escape mechanism is implemented by a micro-subroutine named ESCAPE, which can be called (or rather, jumped to, since ESCAPE never returns to the caller) by any micro-routine that wants to escape to macrocode. ESCAPE is passed the index of the macro-function to be called and from 0 to 2 lisp objects as arguments. (The mechanism of this internal argument-passing is up to the microcode implementor.) ESCAPE then performs the following steps:

1. It is determined where the currently executing macroinstruction is going to place its result, and an appropriate escape-type frame header word is generated.

2. A pointer to the desired function object is fetched from the table of escape functions, as determined by the index that was passed to ESCAPE.

3. The equivalent of a Call instruction is executed for this function object, but the header word determined in step 1 is used instead of the normal header word.

4. The specified arguments, if any, are pushed onto the control stack. The new function is then started by executing the equivalent of a PUSH-LAST instruction.

6.10. Multiple Value Returns

Spice Lisp implements multiple value returns in the style of the Lisp Machine. The macro-instructions needed to implement this feature, and their use, are described here. See also the description of the RETURN instruction in section 6.6.

The CALL-MULTIPLE instruction is used to start a call which is expected to return multiple values. This is identical to the CALL instruction, except that bit 21 of the frame header word is set.

The form (VALUES v1 ... vN) compiles into code that pushes the N values on the stack, then pushes a Misc-Values-Marker on top. This marker has N recorded in its low 24 bits.

When RETURN is called with a CE of type Misc-Values-Marker, it returns from the current function call frame as usual, but instead of pushing a single return value, it pushes the N values it found on the stack, followed by the Mis-Values-Marker from the original CE. See section 6.6 for details.

The net result of performing a multiple-valued call is that N+1 words have been pushed onto the stack: the returned values, in order, with Misc-Values-Marker N on top. Now let us see how this can be used to get the effects of various source constructs.

To do (MULTIPLE-VALUE-LIST (FOO A B)):

```
(CALL-MULTIPLE (CONSTANT [FOO]))
(PUSH [A])
(PUSH-LAST [B])
(MISC %SP-VALUES-TO-N STACK)
(LIST STACK)     ;Pop N from stack, then listify N things.
```

In the MULTIPLE-VALUE call the symbols listed in the first arg are to be set to the respective return values, with missing values defaulting to (). To do (MULTIPLE-VALUE (X Y Z) (FOO A B)):

```
(CALL-MULTIPLE (CONSTANT [FOO]))
(PUSH [A])
(PUSH-LAST [B])
(MISC %SP-VALUES-TO-N STACK)
(- (CONSTANT [3]))      ;Get number offered - number wanted.
(NPOP STACK)            ;Flush surplus returns or push ()s.
(POP [Z])               ;Now put the three values wherever they
(POP [Y])               ; are supposed to go.
(POP [X])
```

In some tail recursive situations, such as in the last form of a PROGN, one function (call it FOO) may want to call another function BAR and return "whatever BAR returns". If BAR returns multiple values, and FOO's caller is expecting multiple values, these values should be passed through. This is done using the CALL-MAYBE-MULTIPLE instruction. Suppose, for example, we are compiling the following:

```
(defun foo (x y)
  (bar x y))
```

This compiles into the following macro-instructions:

```
(call-maybe-multiple (constant bar))
(push (arg x))
(push-last (arg y))
(return stack)
```

If the caller of FOO expects a multiple value, the CALL-MAYBE-MULTIPLE will operate just as a CALL-MULTIPLE does; if the caller of FOO expects a single value, this will operate as a regular CALL.

6.11. Catch and Throw

The Spice Lisp compiler translates (catch tag . expressions) into a call to %sp-catch. This call is opened on the stack, then the folowing arguments are pushed: TAG, CATCHER, THROWP, and PC. This frame is marked on the stack with the %sp-mark-catch-frame misc-op, so that the throw operation can find it. TAG is of course the catch-tag. CATCHER is () for normal calls to catch. THROWP is initially set to (), but is overwritten with the throw-tag if a throw to this tag occurs. The PC argument is a fixnum that indicates where in the calling function's code vector execution should resume after a THROW.

Once these initial arguments have been pushed, the expressions in the catch-form are evaluated as in a progn. It is during the evaluation of these expressions that a throw to this catch may occur. If there is no throw, the last expression may push a value or multiple values the stack. Then the %sp-catch call is started with a push-last.

%Sp-catch itself is a simple macrocoded function that ignores its first four arguments and returns the rest as multiple values. The defintion of %sp-catch (which must be compiled for all of this to work properly) is as follows:

```
(defun %sp-catch (ignore ignore ignore ignore &rest vlist)
  (values-list vlist))
```

A call of the form (CATCH <tag> <expressions>) produces the following code:

```
  (CALL-MAYBE-MULTIPLE %SP-CATCH)       ;Set up the %SP-CATCH call.
  (MISC %SP-MARK-CATCH-FRAME IGNORE)    ;Mark it as a CATCH frame.
  (PUSH <tag>)                          ;Push the arguments.
  (SET-NULL STACK)                      ;Catcher.
  (SET-NULL STACK)                      ;Throwp.
  (PUSH <fixnum return-tag>)            ;PC to go to after throw.
  <code for expressions as progn>    ;Throw may occur anywhere in here.
  (MISC %SP-FLUSH-VALUES IGNORE)     ;Get rid of Misc-Values marker.
  (PUSH-LAST STACK)                  ;If no throw, start %SP-CATCH here.
RETURN-TAG                           ;End up here on throw or no throw.
```

The other catching functions -- catch-all, unwind-all, and unwind-protect have their own internal functions, all of which take the same four initial arguments as %sp-catch. Catch-all and unwind-all always have a TAG of (), which matches any throw-tag, and use the CATCHER argument for passing in the associated function. Unwind-protect is called with a TAG of T, which catches any throw, but only if a matching tag or a tag of () appears higher up on the stack so that the subsequent throw to the same tag will succeed.

The throw macro-op gets a throw-tag, which must be a symbol, as CE and finds a single or multiple value on top of the stack. The intent is that the throw will look up the stack for an open call with the catch-frame marker set and with either a matching tag or a tag of (). If no such tag is found, the THROW signals an error without popping anything off the stack. If a catch-frame with tag T is found, the throw treats it as a match, but only if it finds a truly matching tag or a tag of () higher on the stack; otherwise, an error is signalled as though the throw had found no match at all.

Once a matching open catch-frame is found, the associated call is activated, but with certain of its arguments altered. More precisely, THROW does the following:

1. We need two temporary registers for searching up the stack. Begin with the current Open Frame pointer in register A. Set register B to ().

2. If the header word of frame A has bit 20 set, go to step 4. Otherwise, go on to step 3 to find a new A.

3. If the Previous Open Frame slot of frame A is not (), pick up this value, make it the new A, and return to step 2. If the Previous Open Frame slot of frame A is (), examine the Previous Active Frame of A. If this is (), we have run out of stack and must signal an UNSEEN-THROW-TAG error. If non-(), pick up this value, make it the new A, and repeat step 3 with this new A.

4. Examine the first argument slot of frame A. If this argument is () or is EQ to the tag being thrown to, we have a match. Go on to step 5. If the argument is T and register B is still (), save the pointer to frame A in B. Then return to step 3 and continue searching up the stack.

5. At this point, we have found a match. If register B is non-(), that is the frame we actually want to throw to. Stuff the contents of B into A.

6. Over-write the third argument of frame A with the throw's tag.

7. Shuffle the value or values that were passed to the throw into position as the fifth and subsequent arguments in this frame. Only the values themselves are moved, not the Misc-Values marker. The control stack is then popped so that the last of these values is the top entry on the stack.

8. Unwind the binding stack to the value saved in frame A, restoring the saved values.

9. Begin the call in frame A as though a PUSH-LAST had been done, but first copy the fourth (PC) argument of A into the "saved PC of caller" slot on the stack frame so that catch-function will return to the right place.

6.12. Interrupts and Breaks

There are three kinds of interrupt in Spice Lisp:

Internal Interrupts

On the Perq, and on most other machines on which Spice Lisp might be implemented, certain I/O events must be checked for periodically, and if such an event is detected, some low-level I/O processing must be done. The time constraints are such that we may not be able to wait for a convenient time to do this processing. These are called internal interrputs. The key feature of an internal interrupt is that it is handled by the I/O microcode without altering the state of the Lisp process or its registers in any way. The internal interrupt code may access wired-down pages of main memory, but may not do anything that might cause a page fault or process swap. In some cases (for example, a clock tick) the internal interrupt may set a flag that tells Lisp to take a process break at the next convenient time.

Process Breaks These are more serious. At a process break, the Spice scheduler is given the opportunity to run another process. This may require that the entire state of the Lisp process, including all of its internal registers, be saved in main memory and restored later. On certain machines with insufficient microstore, this may also require that the contents of the control store be swapped, reading in the kernel microcode over the Lisp, and then reading the Lisp back in when it is time to run Lisp again. Process breaks can occur at the time of any kernel call, between any two macro-instructions, and whenever a main memory reference is made which may result in a page fault. At each of these times, the Lisp process must have no state hidden in parts of the micro-machine that cannot be dumped and restored, and the amount of other state should be at a minimum. However, after the break has returned, there is no change in the state of the Lisp process, and processing can go on as though nothing had happened.

Lisp Breaks These are breaks in the flow of processing that are handled within the Lisp process. They can result from errors within Lisp, such things as ↑B breakpoints, etc. Usually, these are handled by doing a Call-Break to some error-handling function within Lisp. This function executes in the current Lisp process and can have Lisp-visible side effects such as changing the value of symbols, altering list structure, etc. If the function does a normal return, its frame disappears from the stack without any trace, and value is returned. The various Lisp breaks that can be encountered at the microcode level are described below, along with the details of how each is handled.

%%% Write the stuff about errors and specific breaks. %%%

Index