Internal Design of CMU Common Lisp on the IBM RT PC

David B. McDonald
Scott E. Fahlman
Skef Wholey

September 1987
CMU-CS-87-157

Abstract

CMU Common Lisp is an implementation of Common Lisp that currently runs on the IBM RT PC under Mach, a Berkeley Unix 4.3 binary compatible operating system. This document describes low level details of the implementation. In particular, it describes the data formats used for all Lisp objects, the assembler language routines (miscops) used to support compiled code, the function call and return mechanism, and other design information necessary to understand the underlying structure of the CMU Common Lisp implementation on the IBM RT PC under the Mach operating system.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), monitored by the Space and Naval Warfare Systems Command under proposed contract N00039-87-C-0251.

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 US Government.

Table of Contents

Acknowledgments

This document is based heavily on the document *Revised Internal Design of Spice Lisp* (`https://www.softwarepreservation.org/projects/LISP/cmu/Spice_Lisp-Revised_Internal_Design-1983.pdf`) by Skef Wholey, Scott Fahlman, and Joseph Ginder.

The FASL file format was designed by Guy L. Steele Jr. and Walter van Roggen, and the appendix on this subject is their document with very few modifications.

1 Introduction

1.1 Scope and Purpose

This document describes a new implementation of CMU Common Lisp (nee Spice Lisp) as it is implemented on the IBM RT PC running Mach, a Berkeley Unix 4.3 binary compatible operating system. This design is undergoing rapid change, and for the present is not guaranteed to accurately describe any past, present, or future implementation of CMU Common Lisp. All questions and comments on this material should be directed to David B. McDonald (David.McDonald@CS.CMU.EDU).

This document specifies the hand-coded assembler routines (miscops) and virtual memory architecture of the IBM RT PC CMU Common Lisp system. This is a working document, and it will change frequently as the system is developed and maintained. If some detail of the system does not agree with what is specified here, it is to be considered a bug.

1.2 Notational Conventions

CMU Common Lisp objects are 32 bits long. The high-order bit of each word is numbered 0; the low-order bit is numbered 31. If a word is broken into smaller units, these are packed into the word from left to right. 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.

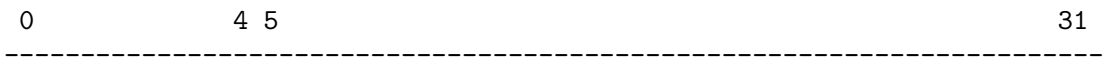
All CMU Common Lisp documentation uses decimal as the default radix; other radices will be indicated by a subscript (as in 77_8) or by a clear statement of what radix is in use.

2 Data Types and Object Formats

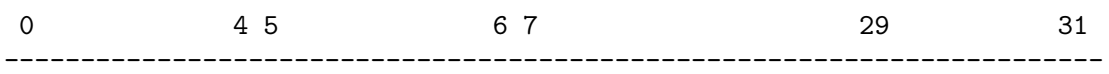
2.1 Lisp Objects

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

Immediate Data Types:



Pointer and Forwarding Types:



2.2 Table of Type Codes

Code	Type	Class	Explanation
0	+ Fixnum	Immediate	Positive fixnum, miscop code, etc.
1	GC-Forward	Pointer	GC forward pointer, used during GC.
4	Bignum	Pointer	Bignum.
5	Ratio	Pointer	Two words: numerator, denominator.
6	+ Short Float	Immediate	Positive short flonum.
7	- Short Float	Immediate	Negative short flonum.
8	Single Float	Pointer	Single precision float.
9	Double Float	Pointer	Double precision float (?).
9	Long Float	Pointer	Long float.
10	Complex	Pointer	Two words: real, imaginary parts.
11	String	Pointer	Character string.
12	Bit-Vector	Pointer	Vector of bits
13	Integer-Vector	Pointer	Vector of integers
14	General-Vector	Pointer	Vector of Lisp objects.
15	Array	Pointer	Array header.
16	Function	Pointer	Compiled function header.
17	Symbol	Pointer	Symbol.
18	List	Pointer	Cons cell.
20	C. S. Pointer	Pointer	Pointer into control stack.
21	B. S. Pointer	Pointer	Pointer into binding stack.
26	Interruptible	Immediate	Marks a miscop as interruptible.
27	Character	Immediate	Character object.
28	Values-Marker	Immediate	Multiple values marker.
29	Catch-All	Immediate	Catch-All object.

```
30      Trap            Immediate       Illegal object trap.
31      - Fixnum        Immediate       Negative fixnum.
```

2.3 Table of Space Codes

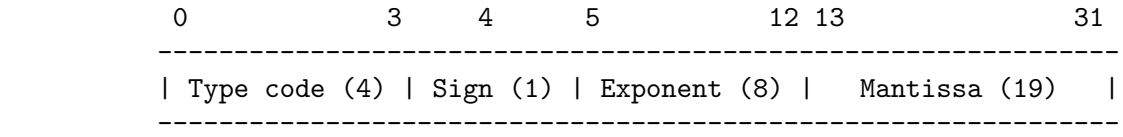
```
Code    Space           Explanation
----    -----           -----------
0       Dynamic-0       Storage normally garbage collected, space 0.
1       Dynamic-1       Storage normally garbage collected, space 1.
2       Static          Permanent objects, never moved or reclaimed.
3       Read-Only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

2.4 Immediate Data Type Descriptions

Fixnum 28-bit two's complement integer. The sign bit is stored redundantly in the top 5 bits of the word.

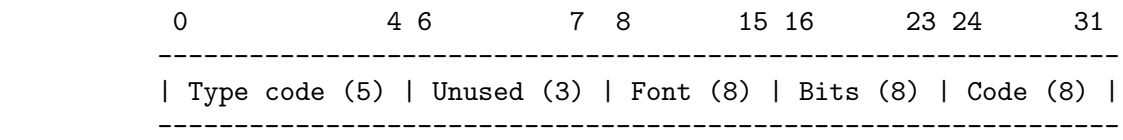
Short-Float

The sign bit is stored as part of the type code, allowing a 28 bit signed short float format. The format of short floating point numbers is:



The floating point number is the same format as the IBM RT PC supports for single precision numbers, except it has been shifted right by four bits for the type code. The result of any operation is therefore truncated. Long floating point numbers are also available if you need more accuracy and better error propagation properties.

Character character object holding a character code, control bits, and font in the following format:



Values-Marker

Used to mark the presence of multiple values on the stack. The low 16 bits indicate how many values are being returned. Note that only 65535 values can be returned from a multiple-values producing form. These are pushed onto the stack in order, and the Values-Marker is returned in register A0.

Catch-All Object used as the catch tag for unwind-protects. Special things happen when a catch frame with this as its tag is encountered during a throw. See [Catch], page 44, for details.

Trap Illegal object trap. This value is used in symbols to signify an undefined value or definition.

Interruptible-Marker
: Object used to mark a miscop as interruptible. This object is put in one of the registers and signals to the interrupt handler that the miscop can be interrupted safely. Only miscops that can take a long time (e.g., length when passed a circular list, system call miscops that may wait indefinitely) are marked this way.

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 Bit-Vectors, Symbols, Lists, and so on. The 5-bit type-code provides the high-order virtual address bits for the object, followed by the 2-bit space code, followed by the 25-bit pointer address. This gives a 30-bit virtual address to a 32-bit word; since the IBM RT PC is a byte-addressed machine, the two low-order bits are 0. In effect we have carved a 30-bit space into a fixed set of 23-bit subspaces, not all of which are used.

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. During a stop and copy garbage collection, a "flip" can be done, turning the old newspace into the new oldspace. All type-spaces are flipped at once. Allocation of new dynamic 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.

Objects in a particular type-space will contain either pointers to garbage-collectible objects or words of raw non-garbage-collectible bits, 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 positive fixnum type-code in the high-order five bits. The remaining three 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 is represented as a fixed-length block of boxed Lisp cells. The number of cells per symbol is 5, in the following order:

```
0  Value cell for shallow binding.
1  Definition cell: a function or list.
2  Property list: a list of attribute-value pairs.
3  Print name: a string.
4  Package: the obarray holding this symbol.
```

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

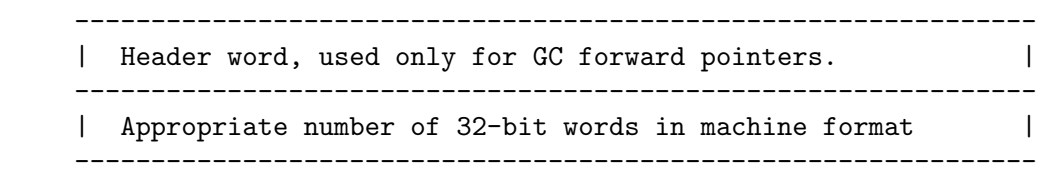
General-Vector
Vector of lisp objects, any length. The first word is a fixnum giving the number of words allocated for the vector (up to 24 bits). The highest legal index is this number minus 2. The second word is vector entry 0, and additional entries are allocated contiguously in virtual memory. General vectors are sometimes called G-Vectors. (See [Vectors], page 8, for further details.)

Integer-Vector
Vector of integers, 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 left-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 low-order end of each word. Integer vectors are sometimes called I-Vectors. (See [Vectors], page 8, for details.)

Bit-Vector Vector of bits, any length. Bit-Vectors are represented in a form identical to I-Vectors, but live in a different space for efficiency reasons.

Bignum Bignums are infinite-precision integers, represented in a format identical to G-Vectors. Each bignum is stored as a series of 32-bit words, with the low-order word stored first. The representation is two's complement, but the sign of the number is redundantly encoded in the type field of the fixnum in the header word. If this fixnum is non-negative, then so is the bignum, if it is negative, so is the bignum.

Floats Floats are stored as two or more consecutive words of bits, in the following format:



The number of words used to represent a floating point number is one plus the size of the floating point number being stored. The floating point numbers will be represented in whatever format the IBM RT PC expects. The extra header word is needed so that a valid floating point number is not mistaken for a gc-forward pointer during a garbage collection.

Ratio Ratios are stored as two consecutive words of Lisp objects, which should both be integers.

Complex Complex numbers are stored as two consecutive words of Lisp objects, which should both be numbers.

Array This is actually a header which holds the accessing and other information about the array. The actual array contents are held in a vector (either an I-Vector

or G-Vector) pointed to by an entry in the header. The header is identical in format to a G-Vector. For details on what the array header contains, see Section 2.8.3 [Arrays], page 10.

String
: A vector of bytes. Identical in form to I-Vectors with the access type always 8-Bit. However, instead of accepting and returning fixnums, string accesses accept and return character objects. 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 CMU Common Lisp function consists of both lisp objects and raw bits for the code. The Lisp objects are stored in the Function space in a format identical to that used for general vectors, with a 24-bit length field in the first word. This object contains assorted parameters needed by the calling machinery, a pointer to an 8-bit I-Vector containing the compiled code, 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.

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 G-Vector space points to a structure in the G-Vector newspace. GC-Forward pointers are only found in oldspace.

2.7 System and Stack Spaces

The virtual addresses below 08000000_{16} are not occupied by Lisp objects, since Lisp objects with type code 0 are positive fixnums. Some of this space is used for other purposes by Lisp. A couple of pages (4096 byte pages) at address 00100000_{16} contain tables that Lisp needs to access frequently. These include the allocation table, the active-catch-frame, information to link to C routines, etc. Memory at location 00200000_{16} contains code for various miscops. Also, any C code loaded into a running Lisp process is loaded after the miscops. The format of the allocation table is described in chapter [Alloc-Chapter], page 19.

The control stack grows upward (toward higher addresses) in memory, and is a framed stack. It contains only general Lisp objects (with some random things encoded as fixnums). 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 binding stack. The frame for a Catch form contains similar information. The precise stack format can be found in chapter [Runtime], page 15.

The special binding stack grows downward. 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.

All Lisp objects are allocated on word boundaries, since the IBM RT PC can only access words on word boundaries.

2.8 Vectors and Arrays

Common Lisp arrays can be represented in a few different ways in CMU Common Lisp – different representations have different performance advantages. Simple general vectors, simple vectors of integers, and simple strings are basic CMU Common Lisp data types, and access to these structures is quicker than access to non-simple (or "complex") arrays. However, all multi-dimensional arrays in CMU Common Lisp are complex arrays, so references to these are always through a header structure.

2.8.1 General Vectors

G-Vectors contain Lisp objects. The format is as follows:

```
------------------------------------------------------------------
|  Fixnum code (5) | Subtype (3) |   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 3-bit subtype field, which is used to indicate several special types of general 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. Normally, 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 using the Shrink-Vector miscop, 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 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 general vector is 2^{24}-2 entries, and that can't fit in the available space. The maximum length is 2^{23}-2 entries, and that is only possible if no other general vectors are present in the space.

Bignums are identical in format to G-Vectors although each entry is a 32-bit integer, and thus only assembler routines should ever access an entry.

Objects of type Function and Array are identical in format to general vectors, though they have their own spaces.

2.8.2 Integer Vectors

I-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 vector. Data going to and from an I-Vector are passed as Fixnums, right justified. Internally these integers are stored in packed form, filling 32-bit words without any type-codes or other overhead. The format is as follows:

```
------------------------------------------------------------------
| Fixnum code (5) | Subtype (3) |  Allocated length (24)       |
------------------------------------------------------------------
| Access type (4) | Number of entries (28)                     |
------------------------------------------------------------------
| Entry 0 left justified                                       |
------------------------------------------------------------------
```

The first word of an I-Vector contains the Fixnum type-code in the top 5 bits, a 3-bit subtype code in the next three 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 vector is the one that is looked at every time the vector is accessed. The low-order 28 bits of this word contain the number of valid entries in the vector, regardless of how long each entry is. The lowest legal index into the 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 Shrink-Vector miscop changes both the allocated length field and the number-of-entries field of an integer 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). The encoding is as follows:

```
0   1-Bit                    8   Unused
1   2-Bit                    9   Unused
2   4-Bit                    10  Unused
3   8-Bit                    11  Unused
4   16-Bit                   12  Unused
5   32-Bit                   13  Unused
6   Unused                   14  Unused
7   Unused                   15  Unused
```

In I-Vectors, the data items are packed into the third and subsequent words of the vector. Item 0 is left justified in the third word, item 1 is to its right, 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 right side of each word. No attempt will be made to split items between words to use up these odd bits. When allocated, an I-Vector is initialized to all 0's.

As with G-Vectors, it is not an error to create an I-Vector of length 0, but it will always be an error to access such a vector. The maximum possible length of an I-Vector is 2^{28}-1 entries or 2^{23}-3 words, whichever is smaller.

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

2.8.3 Arrays

An array header is identical in form to a G-Vector. Like any G-Vector, its first word contains a fixnum type-code, a 3-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 Data Vector
: This is a pointer to the I-Vector, G-Vector, or string 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.

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

2 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.

3 Displacement
: This fixnum value is added to the final code-vector index after the index arithmetic is done but before the access occurs. Used for mapping a portion of one array into another. For most arrays, this is 0.

4 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.

5 - N Ranges of Subsequent Dimensions
: The number of dimensions of an array can be determined by looking at the length of the array header. The rank will be this number minus 6. The maximum array rank is 65535 - 6, or 65529.

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.

2.9 Symbols Known to the Assembler Routines

A large number of symbols will be pre-defined when a CMU Common Lisp system is fired up. A few of these are so fundamental to the operation of the system that their addresses have to be known to the assembler routines. These symbols are listed here. All of these symbols are in static space, so they will not move around.

`NIL` 94000000_{16} The value of NIL is always NIL; it is an error to alter it. The plist of NIL is always NIL; it is an error to alter it. NIL is unique among symbols in that it is stored in Cons cell space and thus you can take its CAR and CDR, yielding NIL in either case. NIL has been placed in Cons cell space so that the more common operations on lists will yield the desired results. This slows down some symbol operations but this should be insignificant compared to the savings in list operations. A test for NIL for the IBM RT PC is:

```
xiu R0,P,X'9400'
bz IsNIL or bnz IsNotNIL
```

`T` $8C000000_{16}$ The value of T is always T; it is an error to alter it. A similar sequence of code as for NIL above can test for T, if necessary.

`%SP-Internal-Apply`
$8C000014_{16}$ The function stored in the definition cell of this symbol is called by an assembler routine whenever compiled code calls an interpreted function.

`%SP-Internal-Error`
$8C000028_{16}$ The function stored in the definition cell of this symbol is called whenever an error is detected during the execution of an assembler routine. See Section 6.5 [Errors], page 45, for details.

`%SP-Software-Interrupt-Handler`
$8C00003C_{16}$ The function stored in the definition cell of this symbol is called whenever a software interrupt occurs. See Section 6.7 [Interrupts], page 49, for details.

`%SP-Internal-Throw-Tag`
$8C000050_{16}$ This symbol is bound to the tag being thrown when a Catch-All frame is encountered on the stack. See [Catch], page 44, for details.

`%Initial-function`
$8c000064_{16}$ This symbol's function cell should contain a function that is called when the initial core image is started. This function should initialize all the data structures that Lisp needs to run.

`%Link-table-header`
$8c000078_{16}$ This symbol's value cell contains a pointer to the link table information.

`Current-allocation-space`
$8c00008c_{16}$ This symbol's value cell contains an encoded form of the current space that new lisp objects are to be allocated in.

`%SP-bignum/fixnum`
$8c0000a0_{16}$ This function is invoked by the miscops when a division of a bignum by a fixnum results in a ratio.

`%SP-bignum/bignum`
: $8c0000b4_{16}$ This function is invoked by the miscops when a division of a fixnum by a bignum results in a ratio.

`%SP-bignum/bignum`
: $8c0000c8_{16}$ This function is invoked by the miscops when a division of a bignum by a bignum results in a ratio.

`%SP-abs-ratio`
: $8c0000dc_{16}$ This function is invoked by the miscops when the absolute value of a ratio is taken.

`%SP-abs-complex`
: $8c0000f0_{16}$ This function is invoked by the miscops when the absolute value of a complex is taken.

`%SP-negate-ratio`
: $8c000104_{16}$ This function is invoked by the miscops when a ratio is to be negated.

`%SP-negate-ratio`
: $8c000118_{16}$ This function is invoked by the miscops when a complex is to be negated.

`%SP-integer+ratio`
: $8c00012c_{16}$ This function is invoked by the miscops when a fixnum or bignum is added to a ratio.

`%SP-ratio+ratio`
: $8c000140_{16}$ This function is invoked by the miscops when a ratio is added to a ratio.

`%SP-complex+number`
: $8c000154_{16}$ This function is invoked by the miscops when a complex is added to a number.

`%SP-number+complex`
: $8c000168_{16}$ This function is invoked by the miscops when a number is added to a complex.

`%SP-complex+complex`
: $8c00017c_{16}$ This function is invoked by the miscops when a number is added to a complex.

`%SP-1+ratio`
: $8c000190_{16}$ This function is invoked by the miscops when 1 is added to a ratio.

`%SP-1+complex`
: $8c000190_{16}$ This function is invoked by the miscops when 1 is added to a complex.

`%SP-ratio-integer`
: $8c0001b8_{16}$ This function is invoked by the miscops when an integer is subtracted from a ratio.

`%SP-ratio-ratio`
: $8c0001cc_{16}$ This function is invoked by the miscops when an ratio is subtracted from a ratio.

`%SP-complex-number`
: $8c0001e0_{16}$ This function is invoked by the miscops when a complex is subtracted from a number.

`%SP-number-complex`
: $8c0001f4_{16}$ This function is invoked by the miscops when a number is subtracted from a complex.

`%SP-complex-complex`
: $8c000208_{16}$ This function is invoked by the miscops when a complex is subtracted from a complex.

`%SP-1-complex`
: $8c000230_{16}$ This function is invoked by the miscops when 1 is subtracted from a complex.

`%SP-ratio*ratio`
: $8c000244_{16}$ This function is invoked by the miscops to multiply two ratios.

`%SP-number*complex`
: $8c000258_{16}$ This function is invoked by the miscops to multiply a number by a complex.

`%SP-complex*number`
: $8c00026c_{16}$ This function is invoked by the miscops to multiply a complex by a number.

`%SP-complex*complex`
: $8c000280_{16}$ This function is invoked by the miscops to multiply a complex by a complex.

`%SP-integer/ratio`
: $8c000294_{16}$ This function is invoked by the miscops to divide an integer by a ratio.

`%SP-ratio/integer`
: $8c0002a8_{16}$ This function is invoked by the miscops to divide a ratio by an integer.

`%SP-ratio/ratio`
: $8c0002bc_{16}$ This function is invoked by the miscops to divide a ratio by a ratio.

`%SP-number/complex`
: $8c0002d0_{16}$ This function is invoked by the miscops to divide a number by a complex.

`%SP-complex/number`
: $8c0002e4_{16}$ This function is invoked by the miscops to divide a complex by a number.

`%SP-complex/complex`
: $8c0002f8_{16}$ This function is invoked by the miscops to divide a complex by a complex.

`%SP-integer-truncate-ratio`
: $8c00030c_{16}$ This function is invoked by the miscops to truncate an integer by a ratio.

`%SP-ratio-truncate-integer`
: $8c000320_{16}$ This function is invoked by the miscops to truncate a ratio by an integer.

`%SP-ratio-truncate-ratio`
: $8c000334_{16}$ This function is invoked by the miscops to truncate a ratio by a ratio.

`%SP-number-truncate-complex`
: $8c000348_{16}$ This function is invoked by the miscops to truncate a number by a complex.

`%SP-complex-truncate-number`
: $8c00035c_{16}$ This function is invoked by the miscops to truncate a complex by a number.

`%SP-complex-truncate-complex`
: $8c000370_{16}$ This function is invoked by the miscops to truncate a complex by a complex.

`Maybe-GC` $8c000384_{16}$ This function may be invoked by any miscop that does allocation. This function determines whether it is time to garbage collect or not. If it is it performs a garbage collection. Whether it invokes a garbage collection or not, it returns the single argument passed to it.

`Lisp-environment-list`
: $8c000398_{16}$ The value of this symbol is set to the a list of the Unix environment strings passed into the Lisp process. This list by Lisp to obtain various environment information, such as the user's home directory, etc.

`Call-lisp-from-C`
: $8c0003ac_{16}$ This function is called whenever a C function called by Lisp tries to call a Lisp function.

`Lisp-command-line-list`
: $8c0003c0_{16}$ The value of this symbol is set to the list of strings passed into the Lisp process as the command line.

`*Nameserverport*`
: $8c0003d4_{16}$ The value of this symbol is set to the C global variable name_server_port. This allows Lisp to access the name server.

`*Ignore-Floating-Point-Underflow*`
: $8c0003e8_{16}$ If the the value of this symbol is NIL then an error is signalled when floating point underflow occurs, otherwise the operation quietly returns zero.

3 Runtime Environment

3.1 Register Allocation

To describe the assembler support routines in chapter [Instr-Chapter], page 21, and the complicated control conventions in chapter [Control-Conventions], page 41, requires that we talk about the allocation of the 16 32-bit general purpose registers provided by the IBM RT PC.

Program-Counter (PC) [R15]
: This register contains an index into the current code vector when a Lisp function is about to be called. When a miscop is called, it contains the return address. It may be used as a super temporary between miscop and function calls.

Active-Function-Pointer (AF) [R14]
: This register contains a pointer to the active function object. It is used to access the symbol and constant area for the currently running function.

Active-Frame-Pointer (FP) [R13]
: This register contains a pointer to the current active frame on the control stack. It is used to access the arguments and local variables stored on the control stack.

Binding-Stack-Pointer (BS) [R12]
: This register contains the current binding stack pointer. The binding stack is a downward growing stack and follows a decrement-write/increment-read discipline.

Local registers (L0-L4) [R7-R11]
: These registers contain locals and saved arguments for the currently executing function. Functions may use these registers, so that stack accesses can be reduced, since a stack access is relatively expensive compared to a register access.

Argument register (A0, A1, A2) [R1, R3, R5]
: These registers contain arguments to a function or miscop that has just been called. On entry to a function or miscop, they contain the first three arguments. The first thing a function does is to move the contents of these registers into the local registers.

Miscop argument register (A3) [R4]
: This register is used to pass a fourth argument to miscops requiring four or more arguments. It is also used as a super temporary by the compiler.

Control-Stack-Pointer (CS) [R6]
: The stack pointer for the control stack, an object of type Control-Stack-Pointer. Points to the last used word in Control-Stack space; this upward growing stack uses a increment-write/read-decrement discipline.

Non-Lisp temporary registers (NL0, NL1) [R0, R2]
: These registers are used to contain non-Lisp values. They will normally be used during miscop calls, but may also be used in in-line code to contain temporary

data. These are the only two registers never examined by the garbage collector, so no pointers to Lisp objects should be stored here (since they won't get updated during a garbage collection).

3.2 Function Object Format

Each compiled function is represented in the machine as a Function Object. This is identical in form to a G-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 G-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 I-Vector that holds the actual code. Following this is the so-called "symbols and constants" area. The first few entries in this area are fixnums that give the offsets into the code vector for various numbers of supplied arguments. Following this begin the true symbols and constants used by the function. Any symbol used by the code as a special variable. Fixnum constants can be generated faster with in-line code than they can be accessed from the function-object, so they are not stored in the constants area.

The subtype of the G-Vector header indicates the type of the function:

- 0 - A normal function (expr).
- 1 - A special form (fexpr).
- 2 - A defmacro macroexpansion function.
- 3 - An anonymous expr. The name is the name of the parent function.
- 4 - A compiled top-level form.

Only the fexpr information has any real meaning to the system. The rest is there for the printer and anyone else who cares.

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

```
0  Name of the innermost enclosing named function.
1  Pointer to the unboxed Code vector holding the instructions.
2  A fixnum with bit fields as follows:
   24  - 31: The minimum legal number of args (0 to 255).
   16  - 23: The maximum number of args, not counting &rest (0 to 255).
         The fixnum has a negative type code, if the function accepts a &rest
         arg and a positive one otherwise.
3  A string describing the source file from which the function was defined.
   See below for a description of the format.
4  A string containing a printed representation of the argument list, for
   documentation purposes.  If the function is a defmacro macroexpansion
   function, the argument list will be the one originally given to defmacro
   rather than the actual arglist to the expansion function.
5  The symbols and constants area starts here.
   This word is entry 0 of the symbol/constant area.
```

```
The first few entries in this area are fixnums representing the
code-vector entry points for various numbers of optional arguments.
```

3.3 Defined-From String Format

The defined-from string may have any of three different formats, depending on which of the three compiling functions compiled it:

compile-file "*filename user-time universal-time*"
: The *filename* is the namestring of the truename of the file the function was defined from. The time is the file-write-date of the file.

compile "Lisp on *user-time*, machine *machine universal-time*"
: The time is the time that the function was compiled. *Machine* is the machine-instance of the machine on which the compilation was done.

compile-from-stream "*stream* on *user-time*, machine *machine-instance universal-time*"
: *Stream* is the printed representation of the stream compiled from. The time is the time the compilation started.

An example of the format of *user-time* is 6-May-86 1:04:44. The *universal-time* is the same time represented as a decimal integer. It should be noted that in each case, the universal time is the last thing in the string.

3.4 Control-Stack Format

The CMU Common Lisp control stack is a framed stack. Call frames, which hold information for function calls, are intermixed with catch frames, which hold information used for non-local exits. In addition, the control stack is used as a scratchpad for random computations.

3.4.1 Call Frames

At any given time, the machine contains pointers to the current top of the control stack and the start of the current active frame (in which the current function is executing). In addition, there is a pointer to the current top of the special binding stack. CMU Common Lisp on the Perq also has a pointer to an open frame. 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 call 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.

On the IBM RT PC, open frames still exist, however, no register is allocated to point at the most recent one. Instead, a count of the arguments to the function is kept. In most cases, a known fixed number of arguments are passed to a function, and this is all that is needed to calculate the correct place to set the active frame pointer. In some cases, it is not as simple, and runtime calculations are necessary to set up the frame pointer. These calculations are simple except in some very strange cases.

The active frame contains pointers to the previously-active 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, stack space is allocated up to the point where the arguments are stored. Nothing is stored in the frame at this time. Thus, as arguments are computed, they can simply be pushed on the stack. Since the first three arguments are passed in registers, it is sometimes necessary to save these values when succeeding arguments are complicated. When the function is finally started, the remainder of the frame is built (including storing all the registers that must be saved). A call frame looks like this:

```
0   Saved local 0 register.
1   Saved local 1 register.
2   Saved local 2 register.
3   Saved local 3 register.
4   Saved local 4 register.
5   Pointer to previous binding stack.
6   Pointer to previous active frame.
7   Pointer to previous active function.
8   Saved PC of caller.  A fixnum.
9   Args-and-locals area starts here.  This is entry 0.
```

The first slot is pointed to by the Active-Frame register if this frame is currently active.

3.4.2 Catch Frames

Catch frames contain much of the same information that call frames do, and have a very similar format. A catch frame holds the function object for the current function, a stack pointer to the current active frame, a pointer to the current top of the binding stack, and a pointer to the previous catch frame. When a Throw occurs, an operation similar to returning from this catch frame (as if it were a call frame) is performed, and the stacks are unwound to the proper place for continued execution in the current function. A catch frame looks like this:

```
0   Pointer to current binding stack.
1   Pointer to current active frame.
2   Pointer to current function object.
3   Destination PC for a Throw.
4   Tag caught by this catch frame.
5   Pointer to previous catch frame.
```

The conventions used to manipulate call and catch frames are described in chapter [Control-Conventions], page 41.

3.5 Binding-Stack Format

Each entry of the binding-stack 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 stack is popped.

4 Storage Management

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. There is also a Clean-Space pointer associated with each space that is used during garbage collection. These pointers are stored in a table which is indexed by the type and space code. The address of the Free-Storage pointer for a given space is

```
(+ alloc-table-base (lsh type 5) (lsh space 3)).
```

The address of the Clean-Space pointer is

```
(+ alloc-table-base (lsh type 5) (lsh space 3) 4).
```

Common Lisp on the IBM RT PC uses a stop-and-copy garbage collector to reclaim storage. The Collect-Garbage miscop performs a full GC. The algorithm used is a degenerate form of Baker's incremental garbage collection scheme. When the Collect-Garbage miscop is executed, the following happens:

1. The current newspace becomes oldspace, and the current oldspace becomes newspace.
2. The newspace Free-Storage and Clean-Space pointers are initialized to point to the beginning of their spaces.
3. The objects pointed at by contents of all the registers containing Lisp objects are transported if necessary.
4. The control stack and binding stack are scavenged.
5. Each static pointer space is scavenged.
6. Each new dynamic space is scavenged. The scavenging of the dynamic spaces continues until an entire pass through all of them does not result in anything being transported. At this point, every live object is in newspace.

A Lisp-level GC function returns the oldspace pages to Mach.

4.1 The Transporter

The transporter moves objects from oldspace to newspace. It is given an address A, which contains the object to be transported, B. If B is an immediate object, a pointer into static space, a pointer into read-only space, or a pointer into newspace, the transporter does nothing.

If B is a pointer into oldspace, the object it points to must be moved. It may, however, already have been moved. Fetch the first word of B, and call it C. If C is a GC-forwarding pointer, we form a new pointer with the type code of B and the low 27 bits of C. Write this into A.

If C is not a GC-forwarding pointer, we must copy the object that B points to. Allocate a new object of the same size in newspace, and copy the contents. Replace C with a GC-forwarding pointer to the new structure, and write the address of the new structure back into A.

Hash tables maintained with an EQ relation need special treatment by the transporter. Whenever a G-Vector with subtype 2 or 3 is transported to newspace, its subtype code

is changed to 4. The Lisp-level hash-table functions will see that the subtype code has changed, and re-hash the entries before any access is made.

4.2 The Scavenger

The scavenger looks through an area of pointers for pointers into oldspace, transporting the objects they point to into newspace. The stacks and static spaces need to be scavenged once, but the new dynamic spaces need to be scavenged repeatedly, since new objects will be allocated while garbage collection is in progress. To keep track of how much a dynamic space has been scavenged, a Clean-Space pointer is maintained. The Clean-Space pointer points to the next word to be scavenged. Each call to the scavenger scavenges the area between the Clean-Space pointer and the Free-Storage pointer. The Clean-Space pointer is then set to the Free-Storage pointer. When all Clean-Space pointers are equal to their Free-Storage pointers, GC is complete.

To maintain (and create) locality of list structures, list space is treated specially. When a list cell is transported, if the cdr points to oldspace, it is immediately transported to newspace. This continues until the end of the list is encountered or a non-oldspace pointer occurs in the cdr position. This linearizes lists in the cdr direction which should improve paging performance.

4.3 Purification

Garbage is created when the files that make up a CMU Common Lisp system are loaded. Many functions are needed only for initialization and bootstrapping (e.g. the "one-shot" functions produced by the compiler for random forms between function definitions), and these can be thrown away once a full system is built. Most of the functions in the system, however, will be used after initialization. Rather than bend over backwards to make the compiler dump some functions in read-only space and others in dynamic space (which involves dumping their constants in the proper spaces, also), *everything* is dumped into dynamic space. A purify miscop is provided that does a garbage collection and moves accessible information in dynamic space into read-only or static space.

5 Assembler Support Routines

To support compiled Common Lisp code many hand coded assembler language routines (miscops) are required. These routines accept arguments in the three argument registers, the special miscop argument register, and in a very few cases on the stack. The current register assignments are:

- A0 contains the first argument.
- A1 contains the second argument.
- A2 contains the third argument.
- A3 contains the fourth argument.

The rest of the arguments are passed on the stack with the last argument at the end of the stack. All arguments on the stack must be popped off the stack by the miscop. All miscops return their values in register A0. A few miscops return two or three values, these are all placed in the argument registers. The main return value is stored in register A0, the others in A1 and A2. The compiler must generate code to use the multiple values correctly, i.e., place the return values on the stack and put a values marker in register A0 if multiple-values are wanted. Otherwise the compiler can use the value(s) it needs and ignore the rest. NB: Most of the miscops follow this scheme, however, a few do not. Any discrepancies are explained in the description of particular miscops.

Several of the instructions described in the Perq Internal Design Document do not have associated miscops, rather they have been code directly in-line. Examples of these instructions include push, pop, bind, bind-null, many of the predicates, and a few other instructions. Most of these instructions can be performed in 4 or fewer IBM RT PC instructions and the overhead of calling a miscop seemed overly expensive. Some instructions are encoded in-line or as a miscop call depending on settings of compiler optimization switches. If space is more important than speed, then some Perq instructions are compiled as calls to out of line miscops rather than generating in-line code.

5.1 Miscop Descriptions

There are 10 classes of miscops: allocation, stack manipulation, list manipulation, symbol manipulation, array manipulation, type predicate, arithmetic and logical, function call and return, miscellaneous, and system hacking.

5.1.1 Allocation

All non-immediate objects are allocated in the "current allocation space," which is dynamic space, static space, or read-only space. The current allocation space is initially dynamic space, but can be changed by using the Set-Allocation-Space miscop below. The current allocation space can be determined by using the Get-Allocation-Space miscop. One usually wants to change the allocation space around some section of code; an unwind protect should be used to insure that the allocation space is restored to some safe value.

Get-Allocation-Space ()
: returns 0, 2, or 3 if the current allocation space is dynamic, static, or read-only, respectively.

Set-Allocation-Space (*X*)
: sets the current allocation space to dynamic, static, or read-only if *X* is 0, 2, or 3 respectively. Returns *X*.

Alloc-Bit-Vector (Length)
: returns a new bit-vector *Length* bits long, which is allocated in the current allocation space. *Length* must be a positive fixnum.

Alloc-I-Vector (*Length A*)
: returns a new I-Vector *Length* bytes long, with the access code specified by *A*. *Length* and *A* must be positive fixnums.

Alloc-String (*Length*)
: returns a new string *Length* characters long. *Length* must be a fixnum.

Alloc-Bignum (*Length*)
: returns a new bignum *Length* 32-bit words long. *Length* must be a fixnum.

Make-Complex (*Realpart Imagpart*)
: returns a new complex number with the specified *Realpart* and *Imagpart*. *Realpart* and *Imagpart* should be the same type of non-complex number.

Make-Ratio (*Numerator Denominator*)
: returns a new ratio with the specified *Numerator* and *Denominator*. *Numerator* and *Denominator* should be integers.

Alloc-G-Vector (*Length Initial-Element*)
: returns a new G-Vector with *Length* elements initialized to *Initial-Element*. *Length* should be a fixnum.

Static-G-Vector (*Length Initial-Element*)
: returns a new G-Vector in static allocation space with *Length* elements initialized to *Initial-Element*.

Vector (Elt_0 Elt_1 ... $Elt_{Length - 1}$ *Length*)
: returns a new G-Vector containing the specified *Length* elements. *Length* should be a fixnum and is passed in register A0. The rest of the arguments are passed on the stack.

Alloc-Function (*Length*)
: returns a new function with *Length* elements. *Length* should be a fixnum.

Alloc-Array (*Length*)
: returns a new array with *Length* elements. *Length* should be a fixnum.

Alloc-Symbol (*Print-Name*)
: returns a new symbol with the print-name as *Print-Name*. The value is initially Trap, the definition is Trap, the property list and the package are initially NIL. The symbol is not interned by this operation – that is done in Lisp code. *Print-Name* should be a simple-string.

Cons (*Car Cdr*)
: returns a new cons with the specified *Car* and *Cdr*.

List (*Elt$_0$ Elt$_1$ ... Elt$_{CE-1}$ Length*)
: returns a new list containing the *Length* elements. *Length* should be fixnum and is passed in register NL0. The first three arguments are passed in A0, A1, and A2. The rest of the arguments are passed on the stack.

List* (*Elt$_0$ Elt$_1$ ... Elt$_{CE-1}$ Length*)
: returns a list* formed by the *Length-1* elements. The last element is placed in the cdr of the last element of the new list formed. *Length* should be a fixnum and is passed in register NL0. The first three arguments are passed in A0, A1, and A2. The rest of the arguments are passed on the stack.

MV-List (*Elt<0> Elt<1> ... Elt<CE - 1> Length*)
: returns a list formed from the elements, all of which are on the stack. *Length* is passed in register A0. This miscop is invoked when multiple values from a function call are formed into a list.

5.1.2 Stack Manipulation

Push (*E*) pushes E on to the control stack.

Pop (*E*) pops the top item on the control stack into *E*.

NPop (*N*) If *N* is positive, *N* items are popped off of the stack. If *N* is negative, NIL is pushed onto the stack -*N* times. *N* must be a fixnum.

Bind-Null (*E*)
: pushes *E* (which must be a symbol) and its current value onto the binding stack, and sets the value of *E* to NIL. Returns NIL.

Bind (Value Symbol)
: pushes *Symbol* (which must be a symbol) and its current value onto the binding stack, and sets the value cell of *Symbol* to *Value*. Returns *Symbol*.

Unbind (*N*)
: undoes the top *N* bindings on the binding stack.

5.1.3 List Manipulation

Car, Cdr, Caar, Cadr, Cdar, Cddr (*E*)
: returns the car, cdr, caar, cadr, cdar, or cddr of *E* respectively.

Set-Cdr, Set-Cddr (*E*)
: The cdr or cddr of the contents of *E* is stored in *E*. The contents of *E* should be either a list or NIL.

Set-Lpop (*E*)
: The car of the contents of *E* is returned; the cdr of the contents of *E* is stored in *E*. The contents of *E* should be a list or NIL.

Spread (*E*)
: pushes the elements of the list *E* onto the stack in left-to-right order.

Replace-Car, Replace-Cdr (*List Value*)
: sets the car or cdr of the *List* to *Value* and returns *Value*.

Endp (X) sets the condition code eq bit to 1 if *X* is NIL, or 0 if *X* is a cons cell. Otherwise an error is signalled.

Assoc, Assq (*List Item*)
returns the first cons in the association-list *List* whose car is EQL to *Item*. If the = part of the EQL comparison bugs out (and it can if the numbers are too complicated), a Lisp-level Assoc function is called with the current cdr of the *List*. Assq returns the first cons in the association-list *List* whose car is EQ to *Item*.

Member, Memq (*List Item*)
returns the first cons in the list *List* whose car is EQL to *Item*. If the = part of the EQL comparison bugs out, a Lisp-level Member function is called with the current cdr of the *List*. Memq returns the first cons in *List* whose car is EQ to the *Item*.

GetF (*List Indicator Default*)
searches for the *Indicator* in the list *List*, cddring down as the Common Lisp form GetF would. If *Indicator* is found, its associated value is returned, otherwise *Default* is returned.

5.1.4 Symbol Manipulation

Most of the symbol manipulation miscops are compiled in-line rather than actual calls.

Get-Value (*Symbol*)
returns the value of *Symbol* (which must be a symbol). An error is signalled if *Symbol* is unbound.

Set-Value (*Symbol Value*)
sets the value cell of the symbol *Symbol* to *Value*. *Value* is returned.

Get-Definition (*Symbol*)
returns the definition of the symbol *Symbol*. If *Symbol* is undefined, an error is signalled.

Set-Definition (*Symbol Definition*)
sets the definition of the symbol *Symbol* to *Definition*. *Definition* is returned.

Get-Plist (*Symbol*)
returns the property list of the symbol *Symbol*.

Set-Plist (*Symbol Plist*)
sets the property list of the symbol *Symbol* to *Plist*. *Plist* is returned.

Get-Pname (*Symbol*)
returns the print name of the symbol *Symbol*.

Get-Package (*Symbol*)
returns the package cell of the symbol *Symbol*.

Set-Package (*Symbol Package*)
sets the package cell of the symbol *Symbol* to *Package*. *Package* is returned.

Boundp (*Symbol*)
: sets the eq condition code bit to 1 if the symbol *Symbol* is bound; sets it to 0 otherwise.

FBoundp (*Symbol*)
: sets the eq condition code bit to 1 if the symbol *Symbol* is defined; sets it to 0 otherwise.

Get (*Symbol Indicator Default*)
: searches the property list of *Symbol* for *Indicator* and returns the associated value. If *Indicator* is not found, *Default* is returned.

Put (*Symbol Indicator Value*)
: searches the property list of *Symbol* for *Indicator* and replaces the associated value with *Value*. If *Indicator* is not found, the *Indicator Value* pair are consed onto the front of the property list.

5.1.5 Array Manipulation

Common Lisp arrays have many manifestations in CMU Common Lisp. The CMU Common Lisp data types Bit-Vector, Integer-Vector, String, General-Vector, and Array are used to implement the collection of data types the Common Lisp manual calls "arrays."

In the following miscop descriptions, "simple-array" means an array implemented in CMU Common Lisp as a Bit-Vector, I-Vector, String, or G-Vector. "Complex-array" means an array implemented as a CMU Common Lisp Array object. "Complex-bit-vector" means a bit-vector implemented as a CMU Common Lisp array; similar remarks apply for "complex-string" and so forth.

Vector-Length (*Vector*)
: returns the length of the one-dimensional Common Lisp array *Vector*. G-Vector-Length, Simple-String-Length, and Simple-Bit-Vector-Length return the lengths of G-Vectors, CMU Common Lisp strings, and CMU Common Lisp Bit-Vectors respectively. *Vector* should be a vector of the appropriate type.

Get-Vector-Subtype (*Vector*)
: returns the subtype field of the vector *Vector* as an integer. *Vector* should be a vector of some sort.

Set-Vector-Subtype (*Vector A*)
: sets the subtype field of the vector *Vector* to *A*, which must be a fixnum.

Get-Vector-Access-Code (*Vector*)
: returns the access code of the I-Vector (or Bit-Vector) *Vector* as a fixnum.

Shrink-Vector (*Vector Length*)
: sets the length field and the number-of-entries field of the vector *Vector* to *Length*. If the vector contains Lisp objects, entries beyond the new end are set to Trap. Returns the shortened vector. *Length* should be a fixnum. One cannot shrink array headers or function headers.

Typed-Vref (*A Vector I*)
: returns the *I*'th element of the I-Vector *Vector* by indexing into it as if its access-code were *A*. *A* and *I* should be fixnums.

Typed-Vset (*A Vector I Value*)
: sets the *I*'th element of the I-Vector *Vector* to *Value* indexing into *Vector* as if its access-code were *A*. *A*, *I*, and *Value* should be fixnums. *Value* is returned.

Header-Length (*Object*)
: returns the number of Lisp objects in the header of the function or array *Object*. This is used to find the number of dimensions of an array or the number of constants in a function.

Header-Ref (*Object I*)
: returns the *I*'th element of the function or array header *Object*. *I* must be a fixnum.

Header-Set (*Object I Value*)
: sets the *I*'th element of the function of array header *Object* to *Value*, and pushes *Value*. *I* must be a fixnum.

The names of the miscops used to reference and set elements of arrays are based somewhat on the Common Lisp function names. The SVref, SBit, and SChar miscops perform the same operation as their Common Lisp namesakes — referencing elements of simple-vectors, simple-bit-vectors, and simple-strings respectively. Aref1 references any kind of one dimensional array. The names of setting functions are derived by replacing "ref" with "set", "char" with "charset", and "bit" with "bitset."

Aref1, SVref, SChar, SBit (*Array I*)
: returns the *I*'th element of the one-dimensional array *Array*. SVref pushes an element of a G-Vector; SChar an element of a string; Sbit an element of a Bit-Vector. *I* should be a fixnum.

Aset1, SVset, SCharset, SBitset (*Array I Value*)
: sets the *I*'th element of the one-dimensional array *Array* to *Value*. SVset sets an element of a G-Vector; SCharset an element of a string; SBitset an element of a Bit-Vector. *I* should be a fixnum and *Value* is returned.

CAref2, CAref3 (*Array I1 I2*)
: returns the element (*I1*, *I2*) of the two-dimensional array *Array*. *I1* and *I2* should be fixnums. CAref3 pushes the element (*I1*, *I2*, *I3*).

CAset2, CAset3 (*Array I1 I2 Value*)
: sets the element (*I1*, *I2*) of the two-dimensional array *Array* to *Value* and returns *Value*. *I1* and *I2* should be fixnums. CAset3 sets the element (*I1*, *I2*, *I3*).

Bit-Bash (*V1 V2 V3 Op*)
: *V1*, *V2*, and *V3* should be bit-vectors and *Op* should be a fixnum. The elements of the bit vector *V3* are filled with the result of *Op*'ing the corresponding elements of *V1* and *V2*. *Op* should be a Boole-style number (see the Boole miscop in section [Boole-Section], page 31).

The rest of the miscops in this section implement special cases of sequence or string operations. Where an operand is referred to as a string, it may actually be an 8-bit I-Vector or system area pointer.

Byte-BLT (*Src-String Src-Start Dst-String Dst-Start Dst-End*)
: moves bytes from *Src-String* into *Dst-String* between *Dst-Start* (inclusive) and *Dst-End* (exclusive). *Dst-Start* - *Dst-End* bytes are moved. If the substrings specified overlap, "the right thing happens," i.e. all the characters are moved to the right place. This miscop corresponds to the Common Lisp function REPLACE when the sequences are simple-strings.

Find-Character (*String Start End Character*)
: searches *String* for the *Character* from *Start* to *End*. If the character is found, the corresponding index into *String* is returned, otherwise NIL is returned. This miscop corresponds to the Common Lisp function FIND when the sequence is a simple-string.

Find-Character-With-Attribute (*String Start End Table Mask*)
: The codes of the characters of *String* from *Start* to *End* are used as indices into the *Table*, which is an I-Vector of 8-bit bytes. When the number picked up from the table bitwise ANDed with *Mask* is non-zero, the current index into the *String* is returned.

SXHash-Simple-String (*String Length*)
: Computes the hash code of the first *Length* characters of *String* and pushes it on the stack. This corresponds to the Common Lisp function SXHASH when the object is a simple-string. The *Length* operand can be Nil, in which case the length of the string is calculated in assembler.

5.1.6 Type Predicates

Many of the miscops described in this sub-section can be coded in-line rather than as miscops. In particular, all the predicates on basic types are coded in-line with default optimization settings in the compiler. Currently, all of these predicates set the eq condition code bit to return an indication of whether the predicate is true or false. This is so that the IBM RT PC branch instructions can be used directly without having to test for NIL. However, this only works if the value of the predicate is needed for a branching decision. In the cases where the value is actually needed, T or NIL is generated in-line according to whether the predicate is true or false. At some point it might be worthwhile having two versions of these predicates, one which sets the eq condition code bit, and one which returns T or NIL. This is especially true if space becomes an issue.

Bit-Vector-P (*Object*)
: sets the eq condition code bit to 1 if *Object* is a Common Lisp bit-vector or 0 if it is not.

Simple-Bit-Vector-P (*Object*)
: sets the eq condition code bit to 1 if *Object* is a CMU Common Lisp bit-vector or 0 if it is not.

Simple-Integer-Vector-P (*Object*)
: sets the eq condition code bit to 1 if *Object* is a CMU Common Lisp I-Vector or 0 if it is not.

StringP (*Object*)
: sets the eq condition code bit to 1 if *Object* is a Common Lisp string or 0 if it is not.

Simple-String-P (*Object*)
: sets the eq condition code bit to 1 if *Object* is a CMU Common Lisp string or 0 if it is not.

BignumP (*Object*)
: sets the eq condition code bit to 1 if *Object* is a bignum or 0 if it is not.

Long-Float-P (*Object*)
: sets the eq condition code bit to 1 if *Object* is a long-float or 0 if it is not.

ComplexP (*Object*)
: sets the eq condition code bit to 1 if *Object* is a complex number or 0 if it is not.

RatioP (*Object*)
: sets the eq condition code bit to 1 if *Object* is a ratio or 0 if it is not.

IntegerP (*Object*)
: sets the eq condition code bit to 1 if *Object* is a fixnum or bignum or 0 if it is not.

RationalP (*Object*)
: sets the eq condition code bit to 1 if *Object* is a fixnum, bignum, or ratio or 0 if it is not.

FloatP (*Object*)
: sets the eq condition code bit to 1 if *Object* is a short-float or long-float or 0 if it is not.

NumberP (*Object*)
: sets the eq condition code bit to 1 if *Object* is a number or 0 if it is not.

General-Vector-P (*Object*)
: sets the eq condition code bit to 1 if *Object* is a Common Lisp general vector or 0 if it is not.

Simple-Vector-P (*Object*)
: sets the eq condition code bit to 1 if *Object* is a CMU Common Lisp G-Vector or 0 if it is not.

Compiled-Function-P (*Object*)
: sets the eq condition code bit to 1 if *Object* is a compiled function or 0 if it is not.

ArrayP (*Object*)
: sets the eq condition code bit to 1 if *Object* is a Common Lisp array or 0 if it is not.

VectorP (*Object*)
: sets the eq condition code bit to 1 if *Object* is a Common Lisp vector of 0 if it is not.

Complex-Array-P (*Object*)
: sets the eq condition code bit to 1 if *Object* is a CMU Common Lisp array or 0 if it is not.

SymbolP (*Object*)
: sets the eq condition code bit to 1 if *Object* is a symbol or 0 if it is not.

ListP (*Object*)
: sets the eq condition code bit to 1 if *Object* is a cons or NIL or 0 if it is not.

ConsP (*Object*)
: sets the eq condition code bit to 1 if *Object* is a cons or 0 if it is not.

FixnumP (*Object*)
: sets the eq condition code bit to 1 if *Object* is a fixnum or 0 if it is not.

Single-Float-P (*Object*)
: sets the eq condition code bit to 1 if *Object* is a single-float or 0 if it is not.

CharacterP (*Object*)
: sets the eq condition code bit to 1 if *Object* is a character or 0 if it is not.

5.1.7 Arithmetic

Integer-Length (*Object*)
: returns the integer-length (as defined in the Common Lisp manual) of the integer *Object*.

Logcount (*Object*)
: returns the number of 1's if *object* is a positive integer, the number of 0's if *object* is a negative integer, and signals an error otherwise.

Float-Short (*Object*)
: returns a short-float corresponding to the number *Object*.

Float-Long (*Number*)
: returns a long float formed by coercing *Number* to a long float. This corresponds to the Common Lisp function Float when given a long float as its second argument.

Realpart (*Number*)
: returns the realpart of the *Number*.

Imagpart (*Number*)
: returns the imagpart of the *Number*.

Numerator (*Number*)
: returns the numerator of the rational *Number*.

Denominator (*Number*)
: returns the denominator of the rational *Number*.

Decode-Float (*Number*)
: performs the Common Lisp Decode-Float function, returning 3 values.

Scale-Float (*Number X*)
: performs the Common Lisp Scale-Float function, returning the result.

= (*X Y*) sets the condition codes according to whether *X* is equal to *Y*. Both *X* and *Y* must be numbers, otherwise an error is signalled. If a rational is compared with a flonum, the rational is converted to a flonum of the same type first. If a short flonum is compared with a long flonum, the short flonum is converted to a long flonum. Flonums must be exactly equal (after conversion) for the condition codes to be set to equality. This miscop also deals with complex numbers.

Compare (*X Y*)
sets the condition codes according to whether *X* is less than, equal to, or greater than *Y*. *X* and *Y* must be numbers. Conversions as described in = above are done as necessary. This miscop replaces the < and > instructions on the Perq, so that the branch on condition instructions can be used more effectively. The value of < and > as defined for the Perq are only generated if necessary, i.e., the result is saved. If *X* or *Y* is a complex number, an error is signalled.

Truncate (*N X*)
performs the Common Lisp TRUNCATE operation. There are 3 cases depending on *X*:

- If *X* is fixnum 1, return two 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 or ratio with the same type as *N*.
- If *X* and *N* are both fixnums or bignums and *X* is not 1, divide *N* by *X*. Return two items: the integer quotient (a fixnum or bignum) and the integer remainder.
- If either *X* or *N* is a flonum or ratio, return a fixnum or bignum quotient (the true quotient rounded toward 0), then a flonum or ratio remainder. The type of the remainder is determined by the same type-coercion rules as for +. The value of the remainder is equal to *N* - *X* * *Quotient*.

On the IBM RT PC, the integer part is returned in register A0, and the remainder in A1.

+, -, *, / (*N X*)
returns *N* + *X*. -, *, and / are similar.

Fixnum*Fixnum, Fixnum/Fixnum (*N X*)
returns *N* * *X*, where both *N* and *X* are fixnums. Fixnum/ is similar.

1+ (*E*) returns *E* + 1.

1- (*E*) returns *E* - 1.

Negate (*N*)
returns -*N*.

Abs (*N*) returns |*N*|.

GCD (*N X*)
returns the greatest common divisor of the integers *N* and *X*.

Logand (*N X*)
returns the bitwise and of the integers *N* and *X*. Logior and Logxor are analogous.

Lognot (*N*)
: returns the bitwise complement of *N*.

Boole (*Op X Y*)
: performs the Common Lisp Boole operation *Op* on *X*, and *Y*. The Boole constants for CMU Common Lisp are these:

```
boole-clr       0
boole-set       1
boole-1         2
boole-2         3
boole-c1        4
boole-c2        5
boole-and       6
boole-ior       7
boole-xor       8
boole-eqv       9
boole-nand      10
boole-nor       11
boole-andc1     12
boole-andc2     13
boole-orc1      14
boole-orc2      15
```

Ash (*N X*)
: performs the Common Lisp ASH operation on *N* and *X*.

Ldb (*S P N*)
: All args are integers; *S* and *P* are non-negative. Performs the Common Lisp LDB operation with *S* and *P* being the size and position of the byte specifier.

Mask-Field (*S P N*)
: performs the Common Lisp Mask-Field operation with *S* and *P* being the size and position of the byte specifier.

Dpb (*V S P N*)
: performs the Common Lisp DPB operation with *S* and *P* being the size and position of the byte specifier.

Deposit-Field (*V S P N*)
: performs the Common Lisp Deposit-Field operation with *S* and *P* as the size and position of the byte specifier.

Lsh (*N X*)
: returns a fixnum that is *N* shifted left by *X* bits, with 0's shifted in on the right. If *X* is negative, *N* is shifted to the right with 0's coming in on the left. Both *N* and *X* should be fixnums.

Logldb (*S P N*)
: All args are fixnums. *S* and *P* specify a "byte" or bit-field of any length within *N*. This is extracted and is returned right-justified as 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. It is an error if the specified field is not entirely within the 26 bits of N

Logdpb (V S P 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. It is an error if the field does not fit into the 26 bits of N.

Sin(X), Cos(X), Tan(X), and Atan(X)
: accept a single number X as argument and return the sine, cosine, tangent, and arctangent of the number respectively. These miscops take advantage of the hardware support provided on the IBM RT PC if it is available, otherwise they escape to Lisp code to calculate the appropriate result.

Log(X) returns the natural log of the number X. This miscop uses the hardware operation if it is available, otherwise it escapes to Lisp code to calculate the result.

Exp(X) returns e raised to the power X. This miscop uses the hardware operation if it is available, otherwise it escapes to Lisp code to calculate the result.

Sqrt(X) returns the square root of X. This miscop uses the hardware operation if it is available, otherwise it escapes to Lisp code to calculate the result.

5.1.8 Branching

All branching is done with IBM RT PC branch instructions. Instructions are generated to set the condition code bits appropriately, and a branch which tests the appropriate condition code bit is generated.

5.1.9 Function Call and Return

Call() A call frame for a function is opened. This is explained in more detail in the next chapter.

Call-0 (F) F must be an executable function, but is a function of 0 arguments. Thus, there is no need to collect arguments. The call frame is opened and activated in a single miscop.

Call-Multiple ()
: Just like a Call miscop, but it marks the frame to indicate that multiple values will be accepted. See section [Multi], page 44.

Set-Up-Apply-Args ()
: is called to handle the last argument of a function called by apply. All the other arguments will have been properly set up by this time. Set-up-apply-args places the values of the list passed as the last argument to apply in their proper locations, whether they belong in argument registers or on the stack. It updates the NArgs register with the actual count of the arguments being passed to the function. When Set-up-apply-args returns, all the arguments to the function being applied are in their correct locations, and the function can be invoked normally.

Start-Call-Interpreter (*NArgs*)
: is called from the interpreter to start a function call. It accepts the number of arguments that are pushed on the stack in register A0. Just below the

arguments is the function to call; just below the function is the area to store the preserved registers. This miscop sets up the argument registers correctly, moves any other arguments down on the stack to their proper place, and invokes the function.

Invoke1 (*Function Argument*)
: is similar to Start-Call-Interpreter, but is simpler, since the *Function* is being called with only a single *Argument*.

Invoke1* (*Function Argument*)
: is similar to Invoke1, but the *Function* being called is called for one value, rather than multiple ones.

Start-call-mc ()
: is called when the compiler generates code for the form multiple-value-call. Register A0 contains the function to be called, A1 contains a 0 if the call if for a single value, and 1 otherwise, NArgs contains the number of arguments that are stored on the stack. The argument registers are set up correctly, and the excess values moved down on the stack if necessary. Finally, the function is actually invoked.

Push-Last ()
: closes the currently open call frame, and initiates a function call.

Return (*X*)
: Return from the current function call. After the current frame is popped off the stack, *X* is returned in register A0 as the result Being returned. See [Return], page 43, for more details.

Return-From (*X F*)
: is similar to Return, except it accepts the frame to return from as an additional argument.

Return-1-Value-Any-Bind (*X*)
: is similar to return, except only one value is returned. Any number of bindings are undone during the return operation.

Return-Mult-Value-0-Bind (*X*)
: is similar to return, except multiple values may be returned, but the binding stack does not have to be popped.

Link-Address-Fixup (*Symbol NArgs Code-Vector Offset*)
: finds the correct link table entry for *Symbol* with *NArgs* (*NArgs* specifies the fixed number of arguments and a flag if more may be passed). It smashes the *Code-Vector* at *Offset* to generate code to point at the absolute address of the link table entry.

Miscop-Fixup (*Code-Vector Offset Index*)
: smashes *Code-Vector* at *Offset* with the correct value for the miscop specified by *Index* in a transfer vector of all the miscops.

Make-Compiled-Closure (*env fcn offset*)
: returns a new function object that is a copy of the function object *fcn* which has the *env* information stored at *offset*. Compiled lexical closures are now

represented as real function objects rather than as lists. This miscop is necessary to support this change.

Reset-link-table (*function*)
: resets all the link table entries for *function* to the default action. This is necessary because Portable Commonloops updates generic function objects by copying new information into the function object. The link table must be updated to reflect this or the wrong function will be called.

Interrupt-Handler (*Signal Code Signal-Context*)
: gets the first indication that a Unix signal has occurred. This miscop does not follow the normal Lisp calling conventions at all. Instead it follows the standard IBM RT PC calling conventions for C or other algorithmic languages. On entry the registers are as follows:

  R0 — Pointer to C data area for Interrupt-Handler. Currently this data area only holds a pointer to the entry point for Interrupt-Handler and nothing else.

  R1 — Pointer to a C stack that contains information about the signal.

  R2 — Contains the *Signal* number that caused the interrupt to happen.

  R3 — Contains the *Code* that further specifies what caused the interrupt (if necessary).

  R4 — Contains a pointer to the *signal-context* which contains information about where the interrupt occurred, the saved registers, etc.

  R5-R14 — Contain unknown values.

  R15 — is the return PC which will return from the interrupt handler and restart the computation.

  Interrupt-Handler determines whether it is safe to take the interrupt now, i.e., it is executing in Lisp code, C code, or an interruptible miscop. An interruptible miscop is one that has been specially written to make sure that it is safe to interrupt it at any point and is possible that it will never return of its own accord (e.g., length which could be passed a circular list, some of the system call miscops, etc.). If it is safe to take the interrupt, the signal-context is modified so that control will transfer to the miscop interrupt-routine when the interrupt-handler returns normally (i.e., after the kernel has done the necessary bookkeeping). If it is unsafe to take the interrupt (i.e., it is executing in an non-interruptible miscop), then the return PC of the miscop is modified to be interrupt-routine and interrupt-handler returns to the kernel. In either case interrupts are disabled and information is stored in a global Lisp data area, so that the interrupt-routine miscop can retrieve the important information about the interrupt.

Interrupt-Routine ()
: gets control when it is safe to take an interrupt. It saves the current state of the computation on the appropriate stack (on the C stack if it was executing in C or on the Lisp stack if in Lisp) including all the registers, some control information

specifying whether the computation was in C code, Lisp code, whether it should form a PC in register R15. When a PC has to be formed in R15, R14 will contain a pointer to the active function and R15 will contain an index into the code vector associated with the active function. Reforming the PC is necessary so it is possible to restart a computation even after a garbage collection may have moved the function. Once this information is stored, interrupt-routine invokes the Lisp function %sp-software-interrupt-routine which moves the processing of the interrupt to Lisp code.

Break-Return ()
: returns from a function called by the interrupt-routine miscop. The only function that should ever do this is %sp-software-interrupt-routine. This miscop expects the stack to be in a format that is generated during an interrupt and should not be used for anything else.

Catch (*Tag PC*)
: builds a catch frame. *Tag* is the tag caught by this catch frame, *PC* is a saved-format PC (i.e., an index into the current code vector). See [Catch], page 44, for details.

Catch-Multiple (*Tag PC*)
: builds a multiple-value catch frame. *Tag* is the tag caught by this catch frame, and *PC* is a saved-format PC. See [Catch], page 44, for details.

Catch-All (*PC*)
: builds a catch frame whose tag is the special Catch-All object. *PC* is the saved-format PC, which is the address to branch to if this frame is thrown through. See [Catch], page 44, for details.

Throw (*X Tag*)
: *Tag* is the throw-tag, normally a symbol. *X* is the value to be returned. See [Catch], page 44, for a description of how this miscop works.

Rest-Entry-0, Rest-Entry-1, Rest-Entry-2, Rest-Entry
: are miscops that do the processing for a function at its &rest entry point. Rest-Entry-*i* are miscops that are invoked by functions that have 0, 1, or 2 arguments before the &rest argument. Rest-entry is invoked for all other cases, and is passed an additional argument in A3 which is the number of non-&rest arguments. These miscops form the &rest arg list and set up all the registers to have the appropriate values. In particular, the non-&rest arguments are copied into preserved registers, and the &rest arg list is built and stored in the appropriate preserved register or on the stack as appropriate.

Call-Foreign (*C-Function Arguments NArgs*)
: establishes the C environment so that C code can be called correctly. *C-Function* is a pointer to the data area for a C function, the first word of which is a pointer to the entry point of the C function. *Arguments* is a block of storage that contains the *NArgs* arguments to be passed to the C function. The first four of these arguments are passed in registers R2 through R5 respectively, the rest are moved onto the C stack in the proper location. When the C function

returns, Call-Foreign restores the Lisp environment and returns as its value the integer in R2.

Call-Lisp (*Arg1 ... Arg2*)
: is a Lisp miscop that gets control when a C function needs to call a Lisp function. Lisp provides a mechanism for setting up an object that looks like a C procedure pointer. The code pointer in this object always points at Call-Lisp. Additional data in this procedure pointer is the Lisp function to call and the number of arguments that it should be called with. Call-Lisp restores the Lisp environment, saves the state of the C computation, moves the C arguments into the correct places for a call to a Lisp function and then invokes the special Lisp function call-lisp-from-c. This Lisp function actually invokes the correct Lisp function. Call-Lisp never regains control.

Return-To-C (*C-Stack-Pointer Value*)
: is used in the function call-lisp-from-c to return control to C from a Lisp function called by C. *C-Stack-Pointer* is the C stack pointer when the call-lisp miscop got control. The C stack pointer argument is used to restore the C environment to what it was at the time the call to Lisp was made. *Value* is the value returned from Lisp and is passed back to C in register R2. Currently, it is not possible to return other than a single 32 bit quantity.

Reset-C-Stack ()
: is invoked when a Lisp function called by C throws out past where it should return to C. Reset-C-Stack restores the C stack to what it was before the original call to C happened. This is so that in the future, the C stack will not contain any garbage that should not be there.

Set-C-Procedure-Pointer (*Sap I Proc*)
: sets the *I/2*'th element of *sap* to be the data part of the statically allocated g-vector *Proc*. This is used to set up a C procedure argument in the argument block that is passed to call-foreign.

5.1.10 Miscellaneous

Eq (X Y) sets the eq condition code bit to 1 if X and Y are the same object, 0 otherwise.

Eql (X Y)
: sets the eq condition code bit to 1 if X and Y are the same object or if X and Y are numbers of the same type with the same value, 0 otherwise.

Make-Predicate (X)
: returns NIL if X is NIL or T if it is not.

Not-Predicate (X)
: returns T if X is NIL or NIL if it is not.

Values-To-N (V)
: V must be a Values-Marker. Returns the number of values indicated in the low 24 bits of V as a fixnum.

N-To-Values (N)
: N is a fixnum. Returns a Values-Marker with the same low-order 24 bits as N.

Force-Values (*VM*)
: If the *VM* is a Values-Marker, do nothing; if not, push *VM* and return a Values-Marker 1.

Flush-Values ()
: is a no-op for the IBM RT PC, since the only time that a Flush-Values miscop is generated is in some well-defined cases where all the values are wanted on the stack.

5.1.11 System Hacking

Get-Type (*Object*)
: returns the five type bits of the *Object* as a fixnum.

Get-Space (*Object*)
: returns the two space bits of *Object* as a fixnum.

Make-Immediate-Type (*X A*)
: returns an object whose type bits are the integer *A* and whose other bits come from the immediate object or pointer *X*. *A* should be an immediate type code.

8bit-System-Ref (*X I*)
: *X* must be a system area pointer, returns the *I*'th byte of *X*, indexing into *X* directly. *I* must be a fixnum.

8bit-System-Set (*X I V*)
: *X* must be a system area pointer, sets the *I*'th element of *X* to *V*, indexing into *X* directly.

16bit-System-Ref (*X I*)
: *X* must be a system area pointer, returns the *I*'th 16-bit word of *X*, indexing into *X* directly.

Signed-16bit-System-Ref (*X I*)
: *X* must be a system area pointer, returns the *I*'th 16-bit word of *X* extending the high order bit as the sign bit.

16bit-System-Set (*X I V*)
: *X* must be a system area pointer, sets the *I*'th element of *X* to *V*, indexing into *X* directly.

Signed-32bit-System-Ref (*X I*)
: *X* must be a system area pointer and *I* an even fixnum, returns the $I/2$'th 32 bit word as a signed quantity.

Unsigned-32bit-System-Ref (*X I*)
: *X* must be a system area pointer and *I* an even fixnum, returns the $I/2$'th 32 bit word as an unsigned quantity.

Signed-32bit-System-Set (*X I V*)
: *X* must be a system area pointer, *I* an even fixnum, and *V* an integer, sets the $I/2$'th element of *X* to *V*.

Sap-System-Ref (X I)
: X must be a system area pointer and I and even fixnum, returns the $I/2$'th element of X as a system area pointer.

Sap-System-Set (X I V)
: X and V must be a system area pointers and I an even fixnum, sets the $I/2$'th element of X to V.

Pointer-System-Set (X I)
: X must be a system area pointer, I an even fixnum, and V a pointer (either system area pointer or Lisp pointer), sets the $I/2$'th element of X to the pointer V. If the pointer is a Lisp pointer, the pointer stored is to the first word of data (i.e., the header word(s) are bypassed).

Sap-Int (X)
: X should be a system area pointer, returns a Lisp integer containing the system area pointer. This miscop is useful when it is necessary to do arithmetic on system area pointers.

Int-Sap (X)
: X should be an integer (fixnum or bignum), returns a system area pointer. This miscop performs the inverse operation of sap-int.

Check-<= (X Y)
: checks to make sure that X is less than or equal to Y. If not, then check-<= signals an error, otherwise it just returns.

Collect-Garbage ()
: causes a stop-and-copy GC to be performed.

Purify ()
: is similar to collect-garbage, except it copies Lisp objects into static or read-only space. This miscop needs Lisp level code to get the process started by putting some root structures into the correct space.

Newspace-Bit ()
: returns 0 if newspace is currently space 0 or 1 if it is 1.

Save (**current-alien-free-pointer** *Checksum memory*)
: Save takes a snap short of the current state of the Lisp computation. The value of the symbol *Current-alien-free-pointer* must be passed to save, so that it can save the static alien data structures. The parameter *checksum* specifies whether a checksum should be generated for the saved image. Currently, this parameter is ignored and no checksum is generated. The parameter *memory* should be be a pointer to a block of memory where the saved core image will be stored. Save returns the size of the core image generated.

Syscall0 Syscall1 Syscall2 Syscall3 Syscall4 Syscall (*number* arg_1 ... arg_n)
: is for making syscalls to the Mach kernel. The argument *number* should be the number of the syscall. Syscall0 accepts no arguments to the syscall; syscall1 accepts one argument to the syscall, etc. Syscall accepts five or more arguments to the syscall.

Unix-Write (*fd buffer offset length*)
: performs a Unix write syscall to the file descriptor *fd*. *Buffer* should contain the data to be written; *Offset* should be an offset into buffer from which to start writing; and *length* is the number of bytes of data to write.

Unix-Fork ()
: performs a Unix fork operation returning one or two values. If an error occurred, the value -1 and the error code is returned. If no error occurred, 0 is returned in the new process and the process id of the child process is returned in the parent process.

Arg-In-Frame (*N F*)
: *N* is a fixnum, *F* is a control stack pointer as returned by the Active-Call-Frame miscop. It returns the item in slot *N* of the args-and-locals area of call frame *F*.

Active-Call-Frame ()
: returns a control-stack pointer to the start of the currently active call frame. This will be of type Control-Stack-Pointer.

Active-Catch-Frame ()
: returns the control-stack pointer to the start of the currently active catch frame. This is Nil if there is no active catch.

Set-Call-Frame (*P*)
: *P* must be a control stack pointer. This becomes the current active call frame pointer.

Current-Stack-Pointer ()
: returns the 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 to the last thing pushed.

Current-Binding-Pointer ()
: returns a Binding-Stack-Pointer that points to the first word above the current top of the binding stack.

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.

Write-Control-Stack (*F V*)
: *F* is a stack pointer, *V* is any Lisp object. Writes *V* into the location addressed. 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 world by using this miscop.

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.

Write-Binding-Stack (B V)

B must be a binding stack pointer. Writes V into the specified location. An error if the location specified is outside the current binding stack.

6 Control Conventions

6.1 Function Calls

On the Perq function calling is done by micro-coded instructions. The instructions perform a large number of operations, including determining whether the function being called is compiled or interpreted, determining that a legal number of arguments are passed, and branching to the correct entry point in the function. To do all this on the IBM RT PC would involve a large amount of computation. In the general case, it is necessary to do all this, but in some common cases, it is possible to short circuit most of this work.

To perform a function call in the general case, the following steps occur:

1. Allocate space on the control stack for the fix-sized part of a call frame. This space will be used to store all the registers that must be preserved across a function call.
2. Arguments to the function are now evaluated. The first three arguments are stored in the argument registers A0, A1, and A2. The rest of the arguments are stored on the stack as they are evaluated. Note that during the evaluation of arguments, the argument registers may be used and may have to be stored in local variables and restored just before the called function is invoked.
3. Load R0 with the argument count.
4. Load the PC register with the offset into the current code vector of the place to return to when the function call is complete.
5. If this call is for multiple values, mark the frame as accepting multiple values, by making the fixnum offset above negative by oring in the negative fixnum type code.
6. Store all the registers that must be preserved over the function call in the current frame.

At this point, all the arguments are set up and all the registers have been saved. All the code to this point is done inline. If the object being called as a function is a symbol, we get the definition from the definition cell of the symbol. If this definition is the trap object, an undefined symbol error is generated. The function calling mechanism diverges at this point depending on the type of function being called, i.e., whether it is a compiled function object or a list.

If we have a compiled function object, the following steps are performed (this code is out of line):

1. Load the active function register with a pointer to the compiled function object.
2. The active frame register is set to the start of the current frame.
3. Note the number of arguments evaluated. Let this be K. The correct entry point in the called function's code vector must be computed as a function of K and the number of arguments the called function wants:
   a. If K < minimum number of arguments, signal an error.
   b. If K > maximum number of arguments and there is no &rest argument, signal an error.
   c. If K > maximum number of arguments and there is a &rest argument, start at offset 0 in the code vector. This entry point must collect the excess arguments into a list and leave the &rest argument in the appropriate argument register or on the stack as appropriate.

   d. If K is between the minimum and maximum arguments (inclusive), get the starting offset from the appropriate slot of the called function's function object. This is stored as a fixnum in slot K - MIN + 6 of the function object.

4. Load one of the Non-Lisp temporary registers with the address of the code vector and add in the offset calculated above. Then do a branch register instruction with this register as the operand. The called function is now executing at the appropriate place.

If the function being called is a list, %SP-Internal-Apply must be called to interpret the function with the given arguments. Proceed as follows:

1. Note the number of arguments evaluated for the current open frame (call this N) and the frame pointer for the frame (call it F). Also remember the lambda expression in this frame (call it L).
2. Load the active function register with the list L.
3. Load the PC register with 0.
4. Allocate a frame on the control stack for the call to %SP-Internal-Apply.
5. Move the contents of the argument registers into the local registers L0, L1, and L2 respectively.
6. Store all the preserved registers in the frame.
7. Place N, F, and L into argument registers A0, A1, and A2 respectively.
8. Do the equivalent of a start call on %SP-Internal-Apply.

%SP-Internal-Apply, a function of three arguments, now evaluates the call to the lambda-expression or interpreted lexical closure L, obtaining the arguments from the frame pointed to by F. The first three arguments must be obtained from the frame that %SP-Internal-Apply runs in, since they are stored in its stack frame and not on the stack as the rest of the arguments are. Prior to returning %SP-Internal-Apply sets the Active-Frame register to F, so that it returns from frame F.

The above is the default calling mechanism. However, much of the overhead can be reduced. Most of the overhead is incurred by having to check the legality of the function call everytime the function is called. In many situations where the function being called is a symbol, this checking can be done only once per call site by introducing a data structure called a link table. The one exception to this rule is when the function apply is used with a symbol. In this situation, the argument count checks are still necessary, but checking for whether the function is a list or compiled function object can be bypassed.

The link table is a hash table whose key is based on the name of the function, the number of arguments supplied to the call and a flag specifying whether the call is done through apply or not. Each entry of the link table consists of two words:

1. The address of the function object associated with the symbol being called. This is here, so that double indirection is not needed to access the function object which must be loaded into the active function register. Initially, the symbol is stored in this slot.
2. The address of the instruction in the function being called to start executing when this table entry is used. Initially, this points to an out of line routine that checks the legality of the call and calculates the correct place to jump to in the called function. This out of line routine replaces the contents of this word with the correct address it calculated. In the case when the call is caused by apply, this will often be an out

of line routine that checks the argument count and calculates where to jump. In the case where the called function accepts &rest arguments and the minimum number of arguments passed is guaranteed to be greater than the maximum number of arguments, then a direct branch to the &rest arg entry point is made.

When a compiled file is loaded into the lisp environment, all the entries for the newly loaded functions will be set to an out of line routine mentioned above. Also, during a garbage collection the entries in this table must be updated when a function object for a symbol is moved.

The IBM RT PC code to perform a function call using the link table becomes:

```
      cal     CS,CS,%Frame-Size       ; Alloc. space on control st.

      <Code to evaluate arguments to the function>

      cau     NL1,0,high-half-word(lte(function nargs flag))
      oil     NL1,0,low-half-word(lte(function nargs flag))
      cal     PC,0,return-tag         ; Offset into code vector.
     <oiu     PC,PC,#xF800            ; Mark if call-multiple frame>
      stm     L0,CS,-(%Frame-Size-4)  ; Save preserved regs.
      lm      AF,NL1,0                ; Link table entry contents.
      bnbrx   pz,R15                  ; Branch to called routine.
      cal     FP,CS,-(%Frame-Size-4)  ; Get pointer to frame.
return-tag:
```

The first two instructions after the arguments are evaled get the address of the link table entry into a register. The two 16-bit half word entries are filled in at load time. The rest of the instructions should be fairly straight forward.

6.2 Returning from a Function Call

Returning from a function call on the Perq is done by a micro-coded instruction. On the IBM RT PC, return has to do the following:

1. Pop the binding stack back to the binding stack pointer stored in the frame we're returning from. For each symbol/value pair popped of the binding stack, restore that value for the symbol.
2. Save the current value of the frame pointer in a temporary registers. This will be used to restore the control stack pointer at the end.
3. Restore all the registers that are preserved across a function call.
4. Get a pointer to the code vector for the function we're returning to. This is retrieved from the code slot of what is now the active function.
5. Make sure the relative PC (which is now in a register) is positive and add it to the code vector pointer above, giving the address of the instruction to return to.
6. If the function is returning multiple values do a block transfer of all the return values down over the stack frame just released, i.e., the first return value should be stored where the temporarily saved frame pointer points to. In effect the return values can be pushed onto the stack using the saved frame pointer above as a stack pointer that is

incremented everytime a value is pushed. Register A0 can be examined to determine the number of values that must be transferred.

7. Set the control stack register to the saved frame pointer above. NB: it may have been updated if multiple values are being returned.
8. Resume execution of the calling function.

Again, it is not always necessary to use the general return code. At compile time it is often possible to determine that no special symbols have to be unbound and/or only one value is being returned. For example the code to perform a return when only one value is returned and it is unnecessary to unbind any special symbols is:

```
cas     NL1,FP,0                ; Save frame register.
lm      L0,FP,0                 ; Restore all preserved regs.
ls      A3,AF,%function-code    ; Get pointer to code vector.
niuo    PC,PC,#x07FF            ; Make relative PC positive.
cas     PC,A3,PC                ; Get addr. of instruction
bnbrx   pz,PC                   ; to return to and do so while
cas     CS,NL1,0                ; updating control stack reg.
```

6.2.1 Returning Multiple-Values

If the current frame can accept multiple values and a values marker is in register A0 indicating N values on top of the stack, it is necessary to copy the N return values down to the top of the control stack after the current frame is popped off. Thus returning multiple values is similar to the above, but a block transfer is necessary to move the returned values down to the correct location on the control stack.

In tail recursive situations, such as in the last form of a PROGN, one function, FOO, may want to call another function, BAR, and return "whatever BAR returns." Call-Multiple is used in this case. If BAR returns multiple values, they will all be passed to FOO. If FOO's caller wants multiple values, the values will be returned. If not, FOO's Return instruction will see that there are multiple values on the stack, but that multiple values will not be accepted by FOO's caller. So Return will return only the first value.

6.3 Non-Local Exits

The Catch and Unwind-Protect special forms are implemented using catch frames. Unwind-Protect builds a catch frame whose tag is the Catch-All object. The Catch miscop creates a catch frame for a given tag and PC to branch to in the current instruction. The Throw miscop looks up the stack by following the chain of catch frames until it finds a frame with a matching tag or a frame with the Catch-All object as its tag. If it finds a frame with a matching tag, that frame is "returned from," and that function is resumed. If it finds a frame with the Catch-All object as its tag, that frame is "returned from," and in addition, %SP-Internal-Throw-Tag is set to the tag being searched for. So that interrupted cleanup forms behave correctly, %SP-Internal-Throw-Tag should be bound to the Catch-All object before the Catch-All frame is built. The protected forms are then executed, and if %SP-Internal-Throw-Tag is not the Catch-All object, its value is thrown to. Exactly what we do is this:

1. Put the contents of the Active-Catch register into a register, A. Put NIL into another register, B.

2. If A is NIL, the tag we seek isn't on the stack. Signal an Unseen-Throw-Tag error.
3. Look at the tag for the catch frame in register A. If it's the tag we're looking for, go to step 4. If it's the Catch-All object and B is NIL, copy A to B. Set A to the previous catch frame and go back to step 2.
4. If B is non-NIL, we need to execute some cleanup forms. Return into B's frame and bind %SP-Internal-Throw-Tag to the tag we're searching for. When the cleanup forms are finished executing, they'll throw to this tag again.
5. If B is NIL, return into this frame, pushing the return value (or BLTing the multiple values if this frame accepts multiple values and there are multiple values).

If no form inside of a Catch results in a Throw, the catch frame needs to be removed from the stack before execution of the function containing the throw is resumed. For now, the value produced by the forms inside the Catch form are thrown to the tag. Some sort of specialized miscop could be used for this, but right now we'll just go with the throw. The branch PC specified by a Catch miscop is part of the constants area of the function object, much like the function's entry points.

6.4 Escaping to Lisp code

Escaping to Lisp code is fairly straight forward. If a miscop discovers that it needs to call a Lisp function, it creates a call frame on the control stack and sets it up so that the called function returns to the function that called the miscop. This means it is impossible to return control to a miscop from a Lisp function.

6.5 Errors

When an error occurs during the execution of a miscop, a call to %SP-Internal-Error is performed. This call is a break-type call, so if the error is proceeded (with a Break-Return instruction), no value will be returned.

%SP-Internal-Error is passed a fixnum error code as its first argument. The second argument is a fixnum offset into the current code vector that points to the location immediately following the instruction that encountered the trouble. From this offset, the Lisp-level error handler can reconstruct the PC of the losing instruction, which is not readily available in the micro-machine. Following the offset, there may be 0 - 2 additional arguments that provide information of possible use to the error handler. For example, an unbound-symbol error will pass the symbol in question as the third arg.

The following error codes are currently defined. Unless otherwise specified, only the error code and the code-vector offset are passed as arguments.

1 Object Not List
: The object is passed as the third argument.

2 Object Not Symbol
: The object is passed as the third argument.

3 Object Not Number
: The object is passed as the third argument.

4 Object Not Integer
: The object is passed as the third argument.

5 Object Not Ratio
: The object is passed as the third argument.

6 Object Not Complex
: The object is passed as the third argument.

7 Object Not Vector
: The object is passed as the third argument.

8 Object Not Simple Vector
: The object is passed as the third argument.

9 Illegal Function Object
: The object is passed as the third argument.

10 Object Not Header
: The object (which is not an array or function header) is passed as the third argument.

11 Object Not I-Vector
: The object is passed as the third argument.

12 Object Not Simple Bit Vector
: The object is passed as the third argument.

13 Object Not Simple String
: The object is passed as the third argument.

14 Object Not Character
: The object is passed as the third argument.

15 Object Not Control Stack Pointer
: The object is passed as the third argument.

16 Object Not Binding Stack Pointer
: The object is passed as the third argument.

17 Object Not Array
: The object is passed as the third argument.

18 Object Not Non-negative Fixnum
: The object is passed as the third argument.

19 Object Not System Area Pointer
: The object is passed as the third argument.

20 Object Not System Pointer
: The object is passed as the third argument.

21 Object Not Float
: The object is passed as the third argument.

22 Object Not Rational
: The object is passed as the third argument.

23 Object Not Non-Complex Number
: A complex number has been passed to the comparison routine for `<` or `>`. The complex number is passed as the third argument.

25 Unbound Symbol
: Attempted access to the special value of an unbound symbol. Passes the symbol as the third argument to %Sp-Internal-Error.

26 Undefined Symbol
: Attempted access to the definition cell of an undefined symbol. Passes the symbol as the third argument to %Sp-Internal-Error.

27 Altering NIL
: Attempt to bind or setq the special value of NIL.

28 Altering T
: Attempt to bind or setq the special value of T.

30 Illegal Vector Access Type
: The specified access type is returned as the third argument.

31 Illegal Vector Size
: Attempt to allocate a vector with negative size or size too large for vectors of this type. Passes the requested size as the third argument.

32 Vector Index Out of Range
: The specified index is out of bounds for this vector. The bad index is passed as the third argument.

33 Illegal Vector Index
: The specified index is not a positive fixnum. The bad index is passed as the third argument.

34 Illegal Shrink Vector Value
: The specified value to shrink a vector to is not a positive fixnum. The bad value is passed as the third argument.

35 Not A Shrink
: The specified value is greater than the current size of the vector being shrunk. The bad value is passed as the third argument.

36 Illegal Data Vector
: The data vector of an array is illegal. The bad vector is passed as the third value.

37 Array has Too Few Indices
: An attempt has been made to access an array as a two or three dimensional array when it has fewer than two or three dimensions, respectively.

38 Array has Too Many Indices
: An attempt has been made to access an array as a two or three dimensional array when it has more than two or three dimensions, respectively.

40 Illegal Byte Specifier
: A bad byte specifier has been passed to one of the byte manipulation miscops. The offending byte specifier is passed as the third argument.

41 Illegal Position in Byte Specifier
: A bad position has been given in a byte specifier that has been passed to one of the byte manipulation miscops. The offending byte specifier is passed as the third argument.

42 Illegal Size in Byte Specifier
: A bad size has been given in a byte specifier that has been passed to one of the byte manipulation miscops. The offending byte specifier is passed as the third argument.

43 Illegal Shift Count
: A shift miscop has encountered non fixnum shift count. The offending shift count is passed as the third argument.

44 Illegal Boole Operation
: The operation code passed to the boole miscop is either not a fixnum or is out of range. The operation code is passed as the third argument.

50 Too Few Arguments
: Too few arguments have been passed to a function. The number of arguments actually passed is passed as the third argument, and the function is passed as the fourth.

51 Too Many Arguments
: Too many arguments have been passed to a function. The number of arguments actually passed is passed as the third argument, and the function is passed as the fourth.

52 Last Apply Arg Not a List
: The last argument to a function being invoked by apply is not a list. The last argument is passed as the third argument.

53 Deleted Link Table Entry
: An attempt has been made to call a function through a link table entry which no longer exists. This is a serious internal error and should never happen.

55 Error Not <=
: The check-<= miscop will invoke this error if the condition is false. The two arguments are passed as the third and fourth arguments to %SP-internal-error.

60 Divide by 0
: An division operation has done a division by zero. The two operands are passed as the third and fourth arguments.

61 Unseen Throw Tag
: An attempt has been made to throw to a tag that is not in the current catch hierarchy. The offending tag is passed as the third argument.

62 Short Float Underflow
: A short float operation has resulted in underflow. The two arguments to the operation are passed as the third and fourth arguments.

63 Short Float Overflow
: A short float operation has resulted in overflow. The two arguments to the operation are passed as the third and fourth arguments.

64 Single Float Underflow
: A single float operation has resulted in underflow. The two arguments to the operation are passed as the third and fourth arguments.

65 Single Float Overflow
: A single float operation has resulted in overflow. The two arguments to the operation are passed as the third and fourth arguments.

66 Long Float Underflow
: A long float operation has resulted in underflow. The two arguments to the operation are passed as the third and fourth arguments.

67 Long Float Overflow
: A long float operation has resulted in overflow. The two arguments to the operation are passed as the third and fourth arguments.

68 Monadic Short Float Underflow
: A short float operation has resulted in underflow. The argument to the operation is passed as the third argument.

69 Monadic Short Float Overflow
: A short float operation has resulted in overflow. The argument to the operation is passed as the third argument.

70 Monadic Long Float Underflow
: A long float operation has resulted in underflow. The argument to the operation is passed as the third argument.

71 Monadic Long Float Overflow
: A long float operation has resulted in overflow. The argument to the operation is passed as the third argument.

6.6 Trapping to the Mach Kernel

Trapping to the Mach kernel is done through one of the syscall0, syscall1, syscall2, syscall3, syscall4, or syscall miscops. The first argument to these miscops is the number of the Unix syscall that is to be invoked. Any other arguments the syscall requires are passed in order after the first one. Syscall0 accepts only the syscall number and no other arguments; syscall1 accepts the syscall number and a single argument to the syscall; etc. Syscall accepts the syscall number and five or more arguments to the Unix syscall. These syscalls generally return two values: the result twice if the syscall succeeded and a -1 and the Unix error code if the syscall failed.

6.7 Interrupts

An interface has been built to the general signal mechanism defined by the Unix operating system. As mentioned in the section on function call and return miscops, several miscops are defined that support the lowest level interface to the Unix signal mechanism. The manual *CMU Common Lisp User's Manual, Mach/IBM RT PC Edition* contains descriptions of functions that allow a user to set up interrupt handlers for any of the Unix signals from within Lisp.

Appendix A Fasload File Format

A.1 General

The purpose of Fasload files is to allow concise storage and rapid loading of Lisp data, particularly function definitions. The intent is that loading a Fasload file has the same effect as loading the ASCII file from which the Fasload file was compiled, but accomplishes the tasks more efficiently. One noticeable difference, of course, is that function definitions may be in compiled form rather than S-expression form. Another is that Fasload files may specify in what parts of memory the Lisp data should be allocated. For example, constant lists used by compiled code may be regarded as read-only.

In some Lisp implementations, Fasload file formats are designed to allow sharing of code parts of the file, possibly by direct mapping of pages of the file into the address space of a process. This technique produces great performance improvements in a paged time-sharing system. Since the Mach project is to produce a distributed personal-computer network system rather than a time-sharing system, efficiencies of this type are explicitly *not* a goal for the CMU Common Lisp Fasload file format.

On the other hand, CMU Common Lisp is intended to be portable, as it will eventually run on a variety of machines. Therefore an explicit goal is that Fasload files shall be transportable among various implementations, to permit efficient distribution of programs in compiled form. The representations of data objects in Fasload files shall be relatively independent of such considerations as word length, number of type bits, and so on. If two implementations interpret the same macrocode (compiled code format), then Fasload files should be completely compatible. If they do not, then files not containing compiled code (so-called "Fasdump" data files) should still be compatible. While this may lead to a format which is not maximally efficient for a particular implementation, the sacrifice of a small amount of performance is deemed a worthwhile price to pay to achieve portability.

The primary assumption about data format compatibility is that all implementations can support I/O on finite streams of eight-bit bytes. By "finite" we mean that a definite end-of-file point can be detected irrespective of the content of the data stream. A Fasload file will be regarded as such a byte stream.

A.2 Strategy

A Fasload file may be regarded as a human-readable prefix followed by code in a funny little language. When interpreted, this code will cause the construction of the encoded data structures. The virtual machine which interprets this code has a *stack* and a *table*, both initially empty. The table may be thought of as an expandable register file; it is used to remember quantities which are needed more than once. The elements of both the stack and the table are Lisp data objects. Operators of the funny language may take as operands following bytes of the data stream, or items popped from the stack. Results may be pushed back onto the stack or pushed onto the table. The table is an indexable stack that is never popped; it is indexed relative to the base, not the top, so that an item once pushed always has the same index.

More precisely, a Fasload file has the following macroscopic organization. It is a sequence of zero or more groups concatenated together. End-of-file must occur at the end of the last

group. Each group begins with a series of seven-bit ASCII characters terminated by one or more bytes of all ones (FF(16)); this is called the *header*. Following the bytes which terminate the header is the *body*, a stream of bytes in the funny binary language. The body of necessity begins with a byte other than FF(16). The body is terminated by the operation `FOP-END-GROUP`.

The first nine characters of the header must be `"FASL FILE"` in upper-case letters. The rest may be any ASCII text, but by convention it is formatted in a certain way. The header is divided into lines, which are grouped into paragraphs. A paragraph begins with a line which does *not* begin with a space or tab character, and contains all lines up to, but not including, the next such line. The first word of a paragraph, defined to be all characters up to but not including the first space, tab, or end-of-line character, is the *name* of the paragraph. A Fasload file header might look something like this:

```
FASL FILE >SteelesPerq>User>Guy>IoHacks>Pretty-Print.Slisp
Package Pretty-Print
Compiled 31-Mar-1988 09:01:32 by some random luser
Compiler Version 1.6, Lisp Version 3.0.
Functions: INITIALIZE DRIVER HACK HACK1 MUNGE MUNGE1 GAZORCH
           MINGLE MUDDLE PERTURB OVERDRIVE GOBBLE-KEYBOARD
           FRY-USER DROP-DEAD HELP CLEAR-MICROCODE
            %AOS-TRIANGLE %HARASS-READTABLE-MAYBE
Macros:    PUSH POP FROB TWIDDLE
<one or more bytes of FF16>
```

The particular paragraph names and contents shown here are only intended as suggestions.

A.3 Fasload Language

Each operation in the binary Fasload language is an eight-bit (one-byte) opcode. Each has a name beginning with `"FOP-"`. In the following descriptions, the name is followed by operand descriptors. Each descriptor denotes operands that follow the opcode in the input stream. A quantity in parentheses indicates the number of bytes of data from the stream making up the operand. Operands which implicitly come from the stack are noted in the text. The notation "→stack" means that the result is pushed onto the stack; "→table" similarly means that the result is added to the table. A construction like "$n(1)$ *value*(n)" means that first a single byte n is read from the input stream, and this byte specifies how many bytes to read as the operand named *value*. All numeric values are unsigned binary integers unless otherwise specified. Values described as "signed" are in two's-complement form unless otherwise specified. When an integer read from the stream occupies more than one byte, the first byte read is the least significant byte, and the last byte read is the most significant (and contains the sign bit as its high-order bit if the entire integer is signed).

Some of the operations are not necessary, but are rather special cases of or combinations of others. These are included to reduce the size of the file or to speed up important cases. As an example, nearly all strings are less than 256 bytes long, and so a special form of string operation might take a one-byte length rather than a four-byte length. As another example, some implementations may choose to store bits in an array in a left-to-right format within each word, rather than right-to-left. The Fasload file format may

support both formats, with one being significantly more efficient than the other for a given implementation. The compiler for any implementation may generate the more efficient form for that implementation, and yet compatibility can be maintained by requiring all implementations to support both formats in Fasload files.

Measurements are to be made to determine which operation codes are worthwhile; little-used operations may be discarded and new ones added. After a point the definition will be "frozen", meaning that existing operations may not be deleted (though new ones may be added; some operations codes will be reserved for that purpose).

0 `FOP-NOP`

No operation. (This is included because it is recognized that some implementations may benefit from alignment of operands to some operations, for example to 32-bit boundaries. This operation can be used to pad the instruction stream to a desired boundary.)

1 `FOP-POP` → table

One item is popped from the stack and added to the table.

2 `FOP-PUSH` *index*(4) → stack

Item number *index* of the table is pushed onto the stack. The first element of the table is item number zero.

3 `FOP-BYTE-PUSH` *index*(1) → stack

Item number *index* of the table is pushed onto the stack. The first element of the table is item number zero.

4 `FOP-EMPTY-LIST` → stack

The empty list (`()`) is pushed onto the stack.

5 `FOP-TRUTH` → stack

The standard truth value (`T`) is pushed onto the stack.

6 `FOP-SYMBOL-SAVE` *n*(4) *name*(*n*)

→ stack & table

The four-byte operand *n* specifies the length of the print name of a symbol. The name follows, one character per byte, with the first byte of the print name being the first read. The name is interned in the default package, and the resulting symbol is both pushed onto the stack and added to the table.

7 `FOP-SMALL-SYMBOL-SAVE` *n*(1) *name*(*n*) → stack & table

The one-byte operand *n* specifies the length of the print name of a symbol. The name follows, one character per byte, with the first byte of the print name being the first read. The name is interned in the default package, and the resulting symbol is both pushed onto the stack and added to the table.

8 `FOP-SYMBOL-IN-PACKAGE-SAVE` *index*(4) *n*(4) *name*(*n*) → stack & table

The four-byte *index* specifies a package stored in the table. The four-byte operand *n* specifies the length of the print name of a symbol. The name follows, one character per byte, with the first byte of the print name being the first read. The name is interned in the specified package, and the resulting symbol is both pushed onto the stack and added to the table.

9 `FOP-SMALL-SYMBOL-IN-PACKAGE-SAVE` *index*(4) *n*(1) *name*(*n*) → stack & table

The four-byte *index* specifies a package stored in the table. The one-byte operand *n* specifies the length of the print name of a symbol. The name follows, one character per byte, with the first byte of the print name being the first read. The name is interned in the specified package, and the resulting symbol is both pushed onto the stack and added to the table.

10 `FOP-SYMBOL-IN-BYTE-PACKAGE-SAVE` *index*(1) *n*(4) *name*(*n*) → stack & table

The one-byte *index* specifies a package stored in the table. The four-byte operand *n* specifies the length of the print name of a symbol. The name follows, one character per byte, with the first byte of the print name being the first read. The name is interned in the specified package, and the resulting symbol is both pushed onto the stack and added to the table.

11 `FOP-SMALL-SYMBOL-IN-BYTE-PACKAGE-SAVE` *index*(1) *n*(1) *name*(*n*) → stack & table

The one-byte *index* specifies a package stored in the table. The one-byte operand *n* specifies the length of the print name of a symbol. The name follows, one character per byte, with the first byte of the print name being the first read. The name is interned in the specified package, and the resulting symbol is both pushed onto the stack and added to the table.

12 Unused.

13 `FOP-DEFAULT-PACKAGE` *index*(4)

A package stored in the table entry specified by *index* is made the default package for future `FOP-SYMBOL` and `FOP-SMALL-SYMBOL` interning operations. (These package FOPs may change or disappear as the package system is determined.)

14 `FOP-PACKAGE` → table

An item is popped from the stack; it must be a symbol. The package of that name is located and pushed onto the table.

15 `FOP-LIST` *length*(1) → stack

The unsigned operand *length* specifies a number of operands to be popped from the stack. These are made into a list of that length, and the list is pushed onto the stack. The first item popped from the stack becomes the last element of the list, and so on. Hence an iterative loop can start with the empty list and perform "pop an item and cons it onto the list" *length* times. (Lists of length greater than 255 can be made by using `FOP-LIST*` repeatedly.)

16 `FOP-LIST*` *length*(1) → stack

This is like `FOP-LIST` except that the constructed list is terminated not by `()` (the empty list), but by an item popped from the stack before any others are. Therefore *length*+1 items are popped in all. Hence an iterative loop can start with a popped item and perform "pop an item and cons it onto the list" *length*+1 times.

17-24 `FOP-LIST-1`, `FOP-LIST-2`, ..., `FOP-LIST-8`
`FOP-LIST-`*k* is like `FOP-LIST` with a byte containing *k* following it. These exist purely to reduce the size of Fasload files. Measurements need to be made to determine the useful values of *k*.

25-32 `FOP-LIST*-1`, `FOP-LIST*-2`, ..., `FOP-LIST*-8`
`FOP-LIST*-`*k* is like `FOP-LIST*` with a byte containing *k* following it. These exist purely to reduce the size of Fasload files. Measurements need to be made to determine the useful values of *k*.

33 `FOP-INTEGER` *n*(4) *value*(*n*) → stack
A four-byte unsigned operand specifies the number of following bytes. These bytes define the value of a signed integer in two's-complement form. The first byte of the value is the least significant byte.

34 `FOP-SMALL-INTEGER` *n*(1) *value*(*n*) → stack
A one-byte unsigned operand specifies the number of following bytes. These bytes define the value of a signed integer in two's-complement form. The first byte of the value is the least significant byte.

35 `FOP-WORD-INTEGER` *value*(4) → stack
A four-byte signed integer (in the range -2^{31} to 2^{31}-1) follows the operation code. A LISP integer (fixnum or bignum) with that value is constructed and pushed onto the stack.

36 `FOP-BYTE-INTEGER` *value*(1) → stack
A one-byte signed integer (in the range -128 to 127) follows the operation code. A LISP integer (fixnum or bignum) with that value is constructed and pushed onto the stack.

37 `FOP-STRING` *n*(4) *name*(*n*) → stack
The four-byte operand *n* specifies the length of a string to construct. The characters of the string follow, one per byte. The constructed string is pushed onto the stack.

38 `FOP-SMALL-STRING` *n*(1) *name*(*n*) → stack
The one-byte operand *n* specifies the length of a string to construct. The characters of the string follow, one per byte. The constructed string is pushed onto the stack.

39 `FOP-VECTOR` *n*(4) → stack
The four-byte operand *n* specifies the length of a vector of LISP objects to construct. The elements of the vector are popped off the stack; the first one popped becomes the last element of the vector. The constructed vector is pushed onto the stack.

40 `FOP-SMALL-VECTOR` *n*(1) → stack
The one-byte operand *n* specifies the length of a vector of LISP objects to construct. The elements of the vector are popped off the stack; the first one popped becomes the last element of the vector. The constructed vector is pushed onto the stack.

41 `FOP-UNIFORM-VECTOR` $n(4)$ $\rightarrow$ stack

The four-byte operand n specifies the length of a vector of LISP objects to construct. A single item is popped from the stack and used to initialize all elements of the vector. The constructed vector is pushed onto the stack.

42 `FOP-SMALL-UNIFORM-VECTOR` $n(1)$ $\rightarrow$ stack

The one-byte operand n specifies the length of a vector of LISP objects to construct. A single item is popped from the stack and used to initialize all elements of the vector. The constructed vector is pushed onto the stack.

43 `FOP-INT-VECTOR` $n(4)$ *size*(1) *count*(1) *data*(ceiling(n/*count*)ceiling(*size***count*/8)) $\rightarrow$ stack

The four-byte operand n specifies the length of a vector of unsigned integers to be constructed. Each integer is *size* bits big, and are packed in the data stream in sections of *count* apiece. Each section occupies an integral number of bytes. If the bytes of a section are lined up in a row, with the first byte read at the right, and successive bytes placed to the left, with the bits within a byte being arranged so that the low-order bit is to the right, then the integers of the section are successive groups of *size* bits, starting from the right and running across byte boundaries. (In other words, this is a consistent right-to-left convention.) Any bits wasted at the left end of a section are ignored, and any wasted groups in the last section are ignored. It is permitted for the loading implementation to use a vector format providing more precision than is required by *size*. For example, if *size* were 3, it would be permitted to use a vector of 4-bit integers, or even vector of general LISP objects filled with integer LISP objects. However, an implementation is expected to use the most restrictive format that will suffice, and is expected to reconstruct objects identical to those output if the Fasload file was produced by the same implementation. (For the PERQ U-vector formats, one would have *size* an element of {1, 2, 4, 8, 16}, and *count*=32/*size*; words could be read directly into the U-vector. This operation provides a very general format whereby almost any conceivable implementation can output in its preferred packed format, and another can read it meaningfully; by checking at the beginning for good cases, loading can still proceed quickly.) The constructed vector is pushed onto the stack.

44 `FOP-UNIFORM-INT-VECTOR` $n(4)$ *size*(1) *value*(ceiling(*size*/8)) $\rightarrow$ stack

The four-byte operand n specifies the length of a vector of unsigned integers to construct. Each integer is *size* bits big, and is initialized to the value of the operand *value*. The constructed vector is pushed onto the stack.

45 `FOP-FLOAT` $n(1)$ *exponent*(ceiling(n/8)) $m(1)$ *mantissa*(ceiling(m/8)) $\rightarrow$ stack

The first operand n is one unsigned byte, and describes the number of *bits* in the second operand *exponent*, which is a signed integer in two's-complement format. The high-order bits of the last (most significant) byte of *exponent* shall equal the sign bit. Similar remarks apply to m and *mantissa*. The value denoted by these four operands is *mantissa*x$2^{exponent\text{-length}(mantissa)}$. A floating-point number shall be constructed which has this value, and then pushed onto the stack. That floating-point format should be used which is the smallest (most

compact) provided by the implementation which nevertheless provides enough accuracy to represent both the exponent and the mantissa correctly.

46-51 Unused

52 `FOP-ALTER` *index*(1)

Two items are popped from the stack; call the first *newval* and the second *object*. The component of *object* specified by *index* is altered to contain *newval*. The precise operation depends on the type of *object*:

List — A zero *index* means alter the car (perform `RPLACA`), and *index*=1 means alter the cdr (`RPLACD`).

Symbol — By definition these indices have the following meaning, and have nothing to do with the actual representation of symbols in a given implementation:

- 0 — Alter value cell.
- 1 — Alter function cell.
- 2 — Alter property list (!).

Vector (of any kind) — Alter component number *index* of the vector.

String — Alter character number *index* of the string.

53 `FOP-EVAL` → stack

Pop an item from the stack and evaluate it (give it to `EVAL`). Push the result back onto the stack.

54 `FOP-EVAL-FOR-EFFECT`

Pop an item from the stack and evaluate it (give it to `EVAL`). The result is ignored.

55 `FOP-FUNCALL` *nargs*(1) → stack

Pop *nargs*+1 items from the stack and apply the last one popped as a function to all the rest as arguments (the first one popped being the last argument). Push the result back onto the stack.

56 `FOP-FUNCALL-FOR-EFFECT` *nargs*(1)

Pop *nargs*+1 items from the stack and apply the last one popped as a function to all the rest as arguments (the first one popped being the last argument). The result is ignored.

57 `FOP-CODE-FORMAT` *id*(1)

The operand *id* is a unique identifier specifying the format for following code objects. The operations `FOP-CODE` and its relatives may not occur in a group until after `FOP-CODE-FORMAT` has appeared; there is no default format. This is provided so that several compiled code formats may co-exist in a file, and so that a loader can determine whether or not code was compiled by the correct compiler for the implementation being loaded into. So far the following code format identifiers are defined:

- 0 — PERQ

1 VAX

3 IBM RT PC

58 `FOP-CODE` *nitems*(4) *size*(4) *code*(*size*) → stack
A compiled function is constructed and pushed onto the stack. This object is in the format specified by the most recent occurrence of `FOP-CODE-FORMAT`. The operand *nitems* specifies a number of items to pop off the stack to use in the "boxed storage" section. The operand *code* is a string of bytes constituting the compiled executable code.

59 `FOP-SMALL-CODE` *nitems*(1) *size*(2) *code*(*size*) → stack
A compiled function is constructed and pushed onto the stack. This object is in the format specified by the most recent occurrence of `FOP-CODE-FORMAT`. The operand *nitems* specifies a number of items to pop off the stack to use in the "boxed storage" section. The operand *code* is a string of bytes constituting the compiled executable code.

60 `FOP-STATIC-HEAP`
Until further notice operations which allocate data structures may allocate them in the static area rather than the dynamic area. (The default area for allocation is the dynamic area; this default is reset whenever a new group is begun. This command is of an advisory nature; implementations with no static heap can ignore it.)

61 `FOP-DYNAMIC-HEAP`
Following storage allocation should be in the dynamic area.

62 `FOP-VERIFY-TABLE-SIZE` *size*(4)
If the current size of the table is not equal to *size*, then an inconsistency has been detected. This operation is inserted into a Fasload file purely for error-checking purposes. It is good practice for a compiler to output this at least at the end of every group, if not more often.

63 `FOP-VERIFY-EMPTY-STACK`
If the stack is not currently empty, then an inconsistency has been detected. This operation is inserted into a Fasload file purely for error-checking purposes. It is good practice for a compiler to output this at least at the end of every group, if not more often.

64 `FOP-END-GROUP`
This is the last operation of a group. If this is not the last byte of the file, then a new group follows; the next nine bytes must be `"FASL FILE"`.

65 `FOP-POP-FOR-EFFECT` stack →
One item is popped from the stack.

66 `FOP-MISC-TRAP` → stack
A trap object is pushed onto the stack.

67 `FOP-READ-ONLY-HEAP`
Following storage allocation may be in a read-only heap. (For symbols, the symbol itself may not be in a read-only area, but its print name (a string) may

be. This command is of an advisory nature; implementations with no read-only heap can ignore it, or use a static heap.)

68 `FOP-CHARACTER` *character*(3) → stack
The three bytes specify the 24 bits of a CMU Common Lisp character object. The bytes, lowest first, represent the code, control, and font bits. A character is constructed and pushed onto the stack.

69 `FOP-SHORT-CHARACTER` *character*(1) → stack
The one byte specifies the lower eight bits of a CMU Common Lisp character object (the code). A character is constructed with zero control and zero font attributes and pushed onto the stack.

70 `FOP-RATIO` → stack
Creates a ratio from two integers popped from the stack. The denominator is popped first, the numerator second.

71 `FOP-COMPLEX` → stack
Creates a complex number from two numbers popped from the stack. The imaginary part is popped first, the real part second.

72 `FOP-LINK-ADDRESS-FIXUP` *nargs*(1) *restp*(1) *offset*(4) → stack
Valid only for when FOP-CODE-FORMAT corresponds to the Vax or the IBM RT PC. This operation pops a symbol and a code object from the stack and pushes a modified code object back onto the stack according to the needs of the runtime code linker on the Vax or IBM RT PC.

73 `FOP-LINK-FUNCTION-FIXUP` *offset*(4) → stack
Valid only for when FOP-CODE-FORMAT corresponds to the Vax or the IBM RT PC. This operation pops a symbol and a code object from the stack and pushes a modified code object back onto the stack according to the needs of the runtime code linker on the Vax or the IBM RT PC.

74 `FOP-FSET`
Pops the top two things off of the stack and uses them as arguments to FSET (i.e. SETF of SYMBOL-FUNCTION).

128 `FOP-LINK-ADDRESS-FIXUP` *nargs* *flag* *offset*
Valid only when FOP-CODE-FORMAT corresponds to the IBM RT PC. This operation pops a symbol and a function object off the stack. The code vector in the function object is modified according to the needs of the runtime code linker of the IBM RT PC and pushed back on the stack. This FOP links in calls to other functions.

129 `FOP-MISCOP-FIXUP` *index*(2) *offset*(4)
Valid only when FOP-CODE-FORMAT corresponds to the IBM RT PC. This operation pops a code object from the stack and pushes a modified code object back onto the stack according to the needs of the runtime code linker on the IBM RT PC. This FOP links in calls to the assembler language support routines.

130 `FOP-ASSEMBLER-ROUTINE` *code-length*
Valid only when FOP-CODE-FORMAT corresponds to the IBM RT PC. This operation loads assembler code into the assembler code space of the currently running Lisp.

131 `FOP-FIXUP-MISCOP-ROUTINE`
Valid only when FOP-CODE-FORMAT corresponds to the IBM RT PC. This operation pops a list of external references, a list of external labels defined, the name, and the code address off the stack. This information is saved, so that after everything is loaded, all the external references can be resolved.

132 `FOP-FIXUP-ASSEMBLER-ROUTINE`
is similar to FOP-FIXUP-MISCOP-ROUTINE, except it is for internal assembler routines rather than ones visible to Lisp.

133 `FOP-FIXUP-USER-MISCOP-ROUTINE`
is similar to FOP-FIXUP-MISCOP-ROUTINE, except it is for routines written by users who have an extremely good understanding of the system internals.

134 `FOP-USER-MISCOP-FIXUP` *offset*(4)
is similar to FOP-MISCOP-FIXUP, but is used to link in user defined miscops.

255 `FOP-END-HEADER`
Indicates the end of a group header, as described above.

Appendix B Building CMU Common Lisp

B.1 Introduction

This document explains how to build a working Common Lisp from source code on the IBM RT PC under the Mach operating system. You should already have a working Common Lisp running on an IBM RT PC before trying to build a new Common Lisp.

Throughout this document the following terms are used:

Core file
: A core file is a file containing an image of a Lisp system. The core file contains header information describing where the data in the rest of the file should be placed in memory. There is a simple C program which reads a core file into memory at the correct locations and then jumps to a location determined by the contents of the core file. The C code includes the X window system version 10 release 4 which may be called from Lisp.

Cold core file
: A cold core file contains enough of the Lisp system to make it possible to load in the rest of the code necessary to generate a full Common Lisp. A cold core file is generated by the program Genesis.

Miscops
: Miscops are assembler language routines that are used to support compiled Lisp code. A Lisp macro assembler provides a convenient mechanism for writing these assembler language routines.

Matchmaker
: Matchmaker is a program developed to automatically generate remote procedure call interfaces between programs. Matchmaker accepts a description of a remote procedure call interface and generates code that implements it.

There are many steps required to go from sources to a working Common Lisp system. Each step will be explained in detail in the following sections. It is possible to perform more than one step with one invocation of Lisp. However, I recommend that each step be started with a fresh Lisp. There is some small chance that something done in one step will adversely affect a following step if the same Lisp is used. The scripts for each step assume that you are in the user package which is the default when Lisp first starts up. If you change to some other package, some of these steps may not work correctly.

In many of the following steps, there are lines setting up search lists so that command files know where to find the sources. What I have done is create a init.lisp file that sets up these search lists for me. This file is automatically loaded from the user's home directory (as determined by the **HOME** environment variable) when you start up Lisp. Note that my init.lisp file is included with the sources. You may have to modify it, if you change where the lisp sources are.

B.2 Installing Source Code

With this document, you should also have received a tape cartridge in tar format containing the complete Common Lisp source code. You should create some directory where you want to put the source code. For the following discussion, I will assume that the source code lives

in the directory /usr/lisp. To install the source code on your machine, issue the following commands:

```
cd /usr/lisp
tar xvf <tape device>
```

The first command puts you into the directory where you want the source code, and the second extracts all the files and sub-directories from the tape into the current directory. <Tape device> should be the name of the tape device on your machine, usually /dev/st0.

The following sub-directories will be created by tar:

`bin` contains a single executable file, lisp, which is a C program used to start up Common Lisp.

`clc` contains the Lisp source code for the Common Lisp compiler and assembler.

`code` contains the Lisp source code that corresponds to all the functions, variables, macros, and special forms described in *Common Lisp: The Language* by Guy L. Steele Jr., as well as some Mach specific files.

`hemlock` contains the Lisp source code for Hemlock, an emacs-like editor written completely in Common Lisp.

`icode` contains Matchmaker generated code for interfaces to Inter Process Communication (IPC) routines. This code is used to communicate with other processes using a remote procedure call mechanism. Under Mach, all the facilities provided by Mach beyond the normal Berkeley Unix 4.3 system calls are accessed from Lisp using this IPC mechanism. Currently, the code for the Mach, name server, Lisp typescript, and Lisp eval server interfaces reside in this directory.

`idefs` contains the Matchmaker definition files used to generate the Lisp code in the icode directory.

`lib` contains files needed to run Lisp. The file lisp.core is known as a Lisp core file and is loaded into memory by the lisp program mentioned above in the entry for the bin directory. This file has a format which allows it to be mapped into memory at the correct locations. The files spell-dictionary.text and spell-dictionary.bin are the text and binary form of a dictionary, respectively, used by Hemlock's spelling checker and corrector. The two files hemlock.cursor and hemlock.mask are used by Hemlock when running under the X window system.

`miscops` contains the Lisp assembler source code for all the miscops that support low level Lisp functions, such as storage allocation, complex operations that can not performed in-line, garbage collection, and other operations. These routines are written in assembler, so that they are as efficient as possible. These routines use a very short calling sequence, so calling them is very cheap compared to a normal Lisp function call.

`mm` contains the Lisp source code for the Matchmaker program. This program is used to generate the Lisp source code files in icode from the corresponding matchmaker definitions in idefs.

`pcl` contains the Lisp source code for a version of the Common Lisp Object System (originally Portable Common Loops), an object oriented programming language built on top of Common Lisp.

`X` contains the C object files for X version 10 release 4 C library routines. These are linked with the lisp startup code, so that X is available from Lisp.

`scribe` contains Scribe source and postscript output for the manuals describing various aspects of the CMU Common Lisp implementation.

`demos` contains the Lisp source code for various demonstration programs. This directory contains the Gabriel benchmark set (bmarks.lisp) and a sub-directory containing the Soar program which is also used for benchmarking purposes. There may be other programs and/or sub-directories here that you may look at.

These directories contain source files as well as Lisp object files. This means it is not necessary to go through all the steps to build a new a Common Lisp, only those steps that are affected by a modification to the sources. For example, modifying the compiler will require recompiling everything. Modifying a miscop file should require only reassembling that particular file and rebuilding the cold core file and full core file.

As well as the directories mentioned above, there are also several files contained in the top-level directory. These are:

`init.lisp`
is a Lisp init file I use. This sets up some standard search lists, as well as defines a Hemlock mode for editing miscop source files.

`lisp.c` contains the C code used to start up the lisp core image under Mach.

`lispstart.s`
contains some assembler language code that is invoked by lisp.c to finish the process of starting up the lisp process.

`makefile` contains make definitions for compiling lisp.c and lispstart.s into the lisp program.

`rg` contains some adb commands that can be read into adb while debugging a lisp process. It prints out all the registers, the name of the currently executing Lisp function, and sets an adb variable to the current stack frame which is used by the following file.

`st` contains some adb commands that can be read into adb while debugging a lisp process. It prints out a Lisp stack frame and the name of the function associated with the stack frame. It also updates the adb variable mentioned above to point to the next stack frame. Repeatedly reading this file into adb will produce a backtrace of all the active call frames on the Lisp stack.

`ac` contains some adb commands that print out the current values of the active catch pointer. This points to the head of a list of catch frames that exist on the control stack.

`cs` contains some adb commands that print out the contents of a catch frame. Reading cs into adb several times in a row (after reading ac once) will print out the catch frames in order.

B.3 Compiling the Lisp Startup Program

To compile the lisp start up program, you should be in the top level directory of the sources (/usr/lisp) and type:

```
make lisp
```

This will compile the file lisp.c, assemble the file lispstart.s and produce an executable file lisp. Currently the default location for the lisp core file is /usr/misc/.lisp/lib/lisp.core. If you want to change this default location, edit the file lisp.c and change the line

```
#define COREFILE "/usr/misc/.lisp/lib/lisp.core"
```

to refer to the file where you intend to put the core file.

This step takes a few seconds.

B.4 Assembling Assembler routines

The standard core image includes a Lisp macro assembler. To assemble all the miscops, the following steps should be performed:

```
(compile-file "/usr/lisp/clc/miscasm.lisp")
(load "/usr/lisp/clc/miscasm.fasl")
(setf (search-list "msc:") '("/usr/lisp/miscops/"))
(clc::asm-files)
```

The first line compiles a file that contains a couple of functions used to assemble miscop source files. The second line loads the resulting compiled file into the currently executing core image. The third line defines the msc search list which is used by the function clc::asm-files to locate the miscop sources. The terminal will display information as each file is assembled. For each file a .fasl, a .list, and an .err file will be generated in /usr/lisp/miscops.

This step takes about half an hour.

B.5 Compiling the Compiler

To compile the compiler is simple:

```
(setf (search-list "clc:") '("/usr/lisp/clc/"))
(load "clc:compclc.lisp")
```

The first line just sets up a search list variable clc, so that the file compclc.lisp can find the compiler sources. The terminal will display information as each file is assembled. For each file a .fasl and an .err file will be generated. A log of the compiler output is also displayed on the terminal.

This step takes about forty-five minutes.

B.6 Compiling the Lisp Sources

Compiling the Lisp source code is also easy:

```
(setf (search-list "code:") '("/usr/lisp/code/"))
(load "code:worldcom.lisp")
```

Again, the first line defines a search list variable, so that the file worldcom.lisp can find the Lisp sources. As each file is compiled, the name of the file is printed on the terminal.

For each file a .fasl will be generated. Also, a single error log will be generated in the file code:compile-lisp.log.

This step takes about an hour and a half.

B.7 Compiling Hemlock

Compiling the Hemlock source code is done as follows:

```
(setf (search-list "hem:") '("/usr/lisp/hemlock/"))
(load "hem:ctw.lisp")
```

Again, the first line defines a search list variable, so that ctw.lisp can find the Hemlock sources. As each file is compiled, the name of the file is printed on the terminal. For each file a .fasl will be generated. Also, a single error log will be generated in the file hem:lossage.log.

This step takes about forty-five minutes.

B.8 Compiling Matchmaker

Compiling the matchmaker sources is done as follows:

```
(setf (search-list "mm:") '("/usr/lisp/mm"))
(compile-file "mm:mm.lisp")
(load "mm:mm.fasl")
(compile-mm)
```

The first line sets up a search list, so that the matchmaker sources can be found. The second line compiles the file containing a function for compiling the matchmaker sources. The third line loads the file just compiled, and the final line invokes the function compile-mm which compiles the matchmaker sources. For each file, a .fasl and .err file is generated. Also, a log of the compiler output is printed to the terminal.

This step takes about 15 minutes

B.9 Generating Lisp Source Files from Matchmaker Definition Files

The following sequence of commands is necessary to generate the Lisp files for the Mach interface:

```
(setf (search-list "mm:") '("/usr/lisp/mm/"))
(setf (search-list "idefs:") '("/usr/lisp/idefs/"))
(setf (search-list "icode:") '("/usr/lisp/icode/"))
(setf (search-list "code:") '("/usr/lisp/code/"))
(setf (default-directory) "/usr/lisp/icode/")
(load "code:mm-interfaces.lisp")
```

The first four lines set up search lists for mm (matchmaker sources), idefs (matchmaker interface definition files), icode (Lisp matchmaker interface sources), and code (Lisp code sources). The fifth line changes the current working directory to be /usr/lisp/icode. This is where the output from matchmaker will be placed. And finally, the last line invokes matchmaker on the matchmaker definition files for all the interfaces.

Matchmaker generates three files for each interface XXX:

`XXXdefs.lisp`
: contains constants and record definitions for the interface.

`XXXmsgdefs.lisp`
: contains definitions of offsets to important fields in the messages that are sent to and received from the interface.

`XXXuser.lisp`
: contains code for each remote procedure, that sends a message to the server and receives the reply from the server (if appropriate). Each of these functions returns one or more values. The first value returned is a general return which specifies whether the remote procedure call succeeded or gives an indication of why it failed. Other values may be returned depending on the particular remote procedure. These values are returned using the multiple value mechanism of Common Lisp.

This step takes about five minutes.

B.10 Compiling Matchmaker Generated Lisp Files

To compile the matchmaker generated Lisp files the following steps should be performed:

```
(setf (search-list "code:") '("/usr/lisp/code/"))
(setf (search-list "icode:") '("/usr/lisp/icode/"))
(load "code:comutil.lisp")
```

The first two lines set up search lists for the code and icode directory. The final line loads a command file that compiles the Mach interface definition in the correct order. Note that once the files are compiled, the XXXmsgdefs files are no longer needed. The file /usr/lisp/icode/lossage.log contains a listing of all the error messages generated by the compiler.

This step takes about fifteen minutes.

B.11 Compiling the Common Lisp Object System

To compile the Common Lisp Object System (CLOS) do the following:

```
(setf (search-list "pcl:") '("/usr/lisp/pcl/"))
(rename-package (find-package "CLOS") "OLD-CLOS")
(compile-file "pcl:defsys.lisp")
(load "pcl:defsys.fasl")
(clos::compile-pcl)
```

The first line sets up a search list as usual. The second line renames the CLOS package to be the OLD-CLOS package. This is so that the current version of CLOS doesn't interfere with the compilation process. The third line compiles a file containing some functions for building CLOS. The fourth line loads in the result of the previous compilation. The final line compiles all the CLOS files necessary for a working CLOS system.

The file /usr/lisp/pcl/test.lisp is a file that contains some test functions. To run it through CLOS build a new Lisp and start up a fresh Lisp resulting from the build and do the following:

```
(in-package 'clos)
(compile-file "/usr/lisp/pcl/test.lisp")
(load "/usr/lisp/pcl/test.fasl")
```

This sequence of function calls puts you in the CLOS package, compiles the test file and then loads it. As the test file is loaded, it executes several tests. It will print out a message specifying whether each test passed or failed.

Currently, CLOS is built into the standard core.

This step takes about 30 minutes.

B.12 Compiling Genesis

To compile genesis do the following:

```
(compile-file "/usr/lisp/clc/genesis.lisp")
```

Genesis is used to build a cold core file. Compiling Genesis takes about five minutes.

B.13 Building a Cold Core File

Once all the files have been assembled or compiled as described above, it is necessary to build a cold core file as follows:

```
(setf (search-list "code:") '("/usr/lisp/code/"))
(setf (search-list "icode:") '("/usr/lisp/icode/"))
(setf (search-list "msc:") '("/usr/lisp/miscops/"))
(load "/usr/lisp/clc/genesis.fasl")
(load "code:worldbuild.lisp")
```

The first three lines set up search lists for the code, icode, and miscops subdirectories. The fourth line loads in the program Genesis which builds the cold core file. The last line calls Genesis on a list of the files that are necessary to build the cold core file. As each file is being processed, its name is printed to the terminal. Genesis generates two files: /usr/lisp/ilisp.core and /usr/lisp/lisp.map. Ilisp.core is the cold core file and lisp.map is a file containing the location of all the functions and miscops in the cold core file. Lisp.map is useful for debugging the cold core file.

This step takes from about fifteen minutes.

B.14 Building a Full Common Lisp

The cold core file built above does not contain some of the more useful programs such as the compiler and hemlock. To build these into a core, it is necessary to do the following:

```
lisp -c /usr/lisp/ilisp.core
(in-package "USER")
(load (open "/usr/lisp/code/worldload.lisp"))
```

The first line invokes the lisp startup program specifying the cold core file just built as the core file to load. This cold core file is set up to do a significant amount of initialization and it is quite possible that some bug will occur during this initialization process. After about a minute, you should get a prompt of the form:

```
CMU Common Lisp kernel core image 2.7(?).
[You are in the Lisp Package.]
*
```

The following two lines should then be entered. The first of these puts you into the User package which is the package you should be in when the full core is first started up. It is necessary to add this line, because the current package is rebound while a file is loaded. The last line loads in a file that loads in the compiler, hemlock, and some other files not yet loaded. The open call is **essential** otherwise when the full core is started up, load will try to close the file and probably invalidate memory that is needed. When load is passed a stream, it does not automatically close the stream. With a file name it now does after a recent bug fix. This file prompts for the versions of the Lisp system, the compiler, and hemlock. You should enter versions that make sense for your installation. It then purifies the core image. Up to this point most of the Lisp system has been loaded into dynamic space. Only a few symbols and some other data structures are in static space. The process of purification moves Lisp objects into static and read-only space, leaving very little in dynamic space. Having the Lisp system in static and read-only space reduces the amount of work the garbage collector has to do. Only those objects needed in the final core file are retained. Finally, a new core file is generated and is written to the file /usr/lisp/nlisp.core. Also, the currently running Lisp should go through the default initialization process, finally prompting for input with an asterisk. At this point you have successfully built a new core file containing a complete Common Lisp implementation.

This step takes about thirty minutes.

B.15 Debugging

Debugging Lisp code is much easier with a fully functional Lisp. However, it is quite possible that a change made in the system can cause a bug to happen while running the cold core file. If this happens, it is best to use adb to track down the problem. Unfortunately, the core file (i.e., the remains of a process normally created by Unix when a process dies) generated by such a bug will be of no use. To get some useful information, follow these steps:

1. Look at the file /usr/lisp/lisp.map and find the entry points for the miscop routines error0, error1, and error2. These entry points are used to invoke the Lisp error system from the miscops. Write down the numbers beside these names. They are the addresses (in hex) of where the miscops are located when the cold core file is loaded into memory.
2. Run adb on the lisp file, i.e.:

   ```
   adb lisp
   ```

3. Set a breakpoint at the lispstart entry point:

   ```
   lispstart:b
   ```

4. Start the lisp program running, telling it to use ilisp.core (I'm assuming you're in /usr/lisp):

   ```
   :r -c ilisp.core
   ```

5. After a while, you will hit the lispstart breakpoint. The core file has been mapped into memory, but control is still in the C startup code. At this point, you should enter breakpoints for all the error entry points described above.

6. Continue running the program by typing :c. Shortly after this, the C lisp program will give up control to Lisp proper. Lisp will start doing its initialization and will probably hit one of the error break points. At that point you can look around at the state and try and discover what has gone wrong. Note that the two files rg and st are useful at this point. Also, you should look at the document *Internal Design of Common Lisp on the IBM RT PC* by David B. McDonald, Scott E. Fahlman, and Skef Wholey so that you know the internal data structures.

B.16 Running the Soar Benchmark

To compile the soar benchmark, you should do the following:

```
(compile-file "/usr/lisp/demos/soar/soar.lisp")
```

To run the benchmark, you should start up a fresh Lisp and do the following:

```
(load "/usr/lisp/demos/soar/soar.fasl")
(load "/usr/lisp/demos/soar/default.soar")
(load "/usr/lisp/demos/soar/eight.soar")
(user-select 'first)
(init-soar)
(time (run))
```

The first two lines load in the standard Soar system. The third line loads in information about the eight puzzle which is a standard Soar puzzle that has been run on several different machines. The fourth line sets up the puzzle conditions so that it will select a goal to work on automatically. The fifth line initializes Soar's working memory, etc. The final line is the one that actually runs the benchmark. Soar prints out a fair amount of information as it solves the puzzle. The final state should be numbered 143 when it finishes. The time macro prints out information about information various resources after the eight puzzle has run.

B.17 Summary

I have tried to present sufficient information here to allow anyone to be able to build a Common Lisp system under Mach from the sources. I am sure there are many tricks that I have learned to use to reduce the amount of grief necessary to build a system. My best recommendation is to go slowly. Start by building a system from the sources provided on the tape. Make sure you are comfortable doing that before you try modifying anything.

Some hints on building the system which you may find useful:

- If you change the compiler, you will have to recompile all the sources before the change is reflected in a system. Changing the compiler is probably the most dangerous change you can make, since an error here means that nothing will work. In particular, this is the time you are going to need to get familiar with adb and the internal structure of the Lisp, since a serious error in the compiler will show up during the initialization of the cold core file.
- Changing the miscops should be done with care. They follow a fairly rigid convention and you should understand all the information provided in *Internal Design of Common Lisp on the IBM RT PC* before making any changes to miscops. You will probably need to get familiar with adb to debug some of the changes. Note that this requires

building a new cold core file and a final core file before the change is reflected in the system.

- Changing sources in the code directory should be fairly straight forward. The only time this will cause trouble is if you change something that a lot of files depend on in which case you will have to recompile everything and build a new cold core file and a core file.
- Changing hemlock should have no adverse effect on system integrity.
- If you make a fairly major change, it is a good idea to go through the complete process of building a core file at least two or three times. If things are still working at the end of this, your change is probably correct and shouldn't cause any serious trouble.
- Finally, always keep at least one backup copy of a good core image around. If you build a bad core file over an existing one and can't back up, it is possible that you may not be able to recover from a serious error.

Index

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W