THE INTERLISP VIRTUAL MACHINE:
A STUDY OF ITS DESIGN AND ITS IMPLEMENTATION AS MULTILISP

by

JOHANNES A.G.M. KOOMEN

B.Sc. THE UNIVERSITY OF BRITISH COLUMBIA 1978

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard.

THE UNIVERSITY OF BRITISH COLUMBIA

July, 1980

(c) Johannes A.G.M. Koomen, 1980
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the Head of my Department or by his representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1W5

Date July 22, 1980
Abstract machine definitions have been recognized as convenient and powerful tools for enhancing software portability. One such machine, the Interlisp Virtual Machine, is examined in this thesis. We present the Multilisp System as an implementation of the Virtual Machine and discuss some of the design criteria and difficulties encountered in mapping the Virtual Machine onto a particular environment. On the basis of our experience with Multilisp we indicate several weaknesses of the Virtual Machine which impair its adequacy as a basis for a portable Interlisp System.
# TABLE OF CONTENTS

1. Introduction .................................................. 1

2. The Interlisp System ........................................... 3
   2.1 Origins of Interlisp ....................................... 3
   2.2 The Interlisp Virtual Machine ......................... 6
   2.3 The VM and the rest of Interlisp ...................... 7

3. The Design of the Interlisp Virtual Machine ............. 10
   3.1 Machine Abstraction .................................... 10
   3.2 The VM Abstraction ..................................... 14
   3.3 Data Manipulation in the VM .......................... 16
      3.3.1 List Cells ....................................... 16
      3.3.2 Literal Atoms ................................... 16
      3.3.3 Numbers ......................................... 17
      3.3.4 Strings ......................................... 18
      3.3.5 Functional Objects ............................. 20
      3.3.6 Other Data Types ................................. 21
   3.4 The Run-time Stack ..................................... 22
      3.4.1 A Model of Multiple Environments ............... 22
      3.4.2 The Spaghetti Stack .............................. 25
      3.4.3 Stack Pointers .................................. 27
      3.4.4 Altering the Flow of Control ................... 29
      3.4.5 Blip Fields .................................... 30
      3.4.6 Copying Frames .................................. 31

4. The Implementation of the Virtual Machine ............... 34
   4.1 The Representation of Objects ......................... 35
      4.1.1 List Cells ....................................... 35
      4.1.2 Literal Atoms ................................... 36
      4.1.3 Numbers ......................................... 37
   4.2 Object Memory Management ............................ 38
      4.2.1 Object Typing .................................. 39
      4.2.2 Garbage Collection .............................. 42
   4.3 The Spaghetti Stack .................................... 44
      4.3.1 Stack Segments .................................. 45
      4.3.2 Stack Elements .................................. 46
      4.3.3 Stack Frames ................................... 47
      4.3.4 Temporaries and Blip Fields .................... 49
      4.3.5 Argument Binding ................................. 51
   4.4 The Multilisp Interpreter ............................. 52
      4.4.1 The Basic Loop ................................... 52
      4.4.2 Argument Evaluation .............................. 54
      4.4.3 Function Invocation .............................. 55
      4.4.4 An Example: the Implementation of COND ....... 56
Table of Contents

5. Evaluation ................................................................. 59
   5.1 The Virtual Machine ............................................. 59
       5.1.1 System Dependencies ....................................... 60
       5.1.2 Complexity .................................................. 61
       5.1.3 Redundancy .................................................. 63
       5.1.4 Incompleteness .............................................. 64
   5.2 Multilisp ............................................................... 65
       5.2.1 The Implementation Language .............................. 65
       5.2.2 Multilisp and the VM ....................................... 67
   5.3 Directions for Future Research .................................. 68

6. Conclusion ................................................................. 70

Bibliography ................................................................. 72

Appendix A  SUBR's in the VM and in Interlisp ...................... 76
Appendix B  Object Representation in Multilisp .................... 78
Appendix C  Multilisp List Compaction ............................... 80
Appendix D  Multilisp Stack Frame Layout ............................ 81
Appendix E  Multilisp Interpreter Loop ............................... 84
Appendix F  Annotated Trace of Stack Example ....................... 85
I wish to express my sincere gratitude to my supervisors Dr. Alan Mackworth and Dr. Bary Pollack for their tremendous patience, inspiration and continual encouragement. I am also greatly indebted to Alan for his generous financial support in the form of a Research Assistantship as well as ample computing resources.

I would like to thank Dan Razzell for his excellent assistance in the development of the software, Dr. Bill Havens for our many enlightening discussions and David Lowe for providing some of the basic ideas for the implementation.

I dedicate this thesis to my wife Trisha -- her faith, forbearance and loving support have made it all come true.
1. Introduction

LISP remains today both a much celebrated and highly controversial programming language. Throughout its 20 years of existence, from its conception in the late 1950's by John McCarthy [McCarthy, 1960], who laid the theoretical foundations, to today's multitude of versions, derivatives and dialects, such as MACLISP, QLISP, PLANNER, CONNIVER and INTERLISP, it has remained surprisingly similar in its basic definition. Each of the later systems contains a basic LISP kernel around which an (often elaborate) collection of programming and/or user support facilities has been constructed.

This thesis is a study of the design and implementation of the kernel of Interlisp as defined in "the Interlisp Virtual Machine Specification" [Moore, 1976]. One of our main concerns will be the portability of Interlisp, and to what extend the Virtual Machine promotes it. In Chapter 2 we shall present a brief historic overview of Interlisp, introduce the Interlisp Virtual Machine, and describe in summary how the Interlisp System interacts with the Virtual Machine. The design of the Interlisp Virtual Machine is discussed in Chapter 3, focusing primarily on data specifications and control specifications. Chapter 4 discusses Multilisp, a partial implementation of the Virtual Machine, particularly the difficulties encountered in mapping the Virtual Machine onto a particular environment and some of the design criteria and decisions made in this implementation. An evaluation of the Virtual Machine and of the
implementation is given in Chapter 5, as well as a discussion of some viable directions for future research. We conclude this thesis in Chapter 6 with a summary of our findings.
2. The Interlisp System

2.1 Origins of Interlisp

In 1960 John McCarthy [McCarthy, 1960] published a paper that had a profound influence on the Artificial Intelligence community. This paper presents a formal definition of recursive functions for symbolic expressions and introduces a programming system called LISP embodying this formalism. Two years later, the "LISP 1.5 Programmer's Manual" was published [McCarthy et al., 1962] which to date is still an important and relevant reference for all beginning LISP'ers. The Manual describes the LISP language in a formal Meta-language, defines the operation of the LISP Interpreter, describes added features such as arithmetic, array processing and the PROG feature, and explains how to run the system.

Although strongly batch oriented (cards and tapes for input, print and punch for output), this early system running on the IBM 7090 contains most of what today would be considered the essentials of any LISP system including a compiler and a symbolic representation of Machine Language (LAP). Indeed, even today all LISP systems still incorporate some original LISP 1.5 or IBM 7090 idiosyncrasies!

In 1966 a version of LISP was developed by D.G. Bobrow and D.L. Murphy at Bolt, Beranek & Newman [Teitelman, 1978] which ran on Digital Equipment's PDP-1. This version was essentially the same as the original LISP 1.5.
An upwards compatible version of this LISP was developed by Bobrow and Murphy in 1967, called 940 LISP, for the SDS-940. This version supported many of the features that currently make up Interlisp: compiled code was compatible with interpreted code, extensive debugging facilities were provided through uniform error processing, and the system contained an editor specifically designed for editing symbolic expressions. But most importantly, 940 LISP was the first LISP system that used virtual memory and software paging [Bobrow & Murphy, 1967], thus greatly increasing the amount of freespace available to the LISP programmer.

The DWIM facility, written by W. Teitelman, was first incorporated in 940 LISP in 1968. DWIM (Do What I Mean) is an automatic syntax error correction facility.

A new LISP version, compatible with 940 LISP, was developed by D.G. Bobrow, D.L. Murphy, A.K. Hartley and W. Teitelman in 1970. This version, called BBN LISP, was designed for the PDP-10 and ran under TENEX, a new and sophisticated operating system designed by Bobrow and others [1972].

Freed from previous hardware (memory) constraints, the following years saw a rapid growth of the system. In 1971 a new and improved DWIM was installed. The introduction of CLISP [Teitelman, 1976] provided for a more Algol-like program syntax. The Programmer's Assistant was introduced in 1972. (For a detailed description of these subsytems see [Teitelman, 1978]).
In 1973 Bolt, Beranek & Newman and Xerox Palo Alto Research Center shared in the maintenance and development of BBN LISP. At this time the name Interlisp was introduced, both to indicate the joint venture and to emphasize the highly interactive nature of the system.

The Interlisp Compiler was greatly improved in 1974 through the notion of block compiling, linked function calls and code swapping. Block compiling involves compiling several functions together into one block of code. Function calls within the block are very fast since stack information for each invocation is kept to a minimum, and the function calls themselves are linked. Linking functions involves compiling function calls not by applying the functional identifier but applying directly the contents of the Function Definition Field of the functional identifier. The result of this is that the contents of the Functional Definition Field need not be examined at runtime.

In 1975 the spaghetti stack was incorporated into Interlisp. This stack mechanism, based on the model of multiple stacks as defined by Bobrow and Wegbreit [1973], is described in more detail in Chapter 3.

The system whose evolution has been described briefly here (generally referred to as Interlisp-10) is running at some dozen different locations, on PDP-10's, under TENEX as well as under the TOPS-20 operating system. A number of versions have been, or are in the process of being implemented, with varying
success, at the University of Lynkoping in Sweden, the University of Tel Aviv in Israel, and the University of British Columbia in Vancouver.

2.2 The Interlisp Virtual Machine

In 1976 Xerox Palo Alto Research Center published "The Interlisp Virtual Machine Specification" [Moore, 1976]. The Interlisp Virtual Machine (henceforth referred to as "the VM") was introduced in order to ensure compatibility between versions and with other implementations. It is an attempt to capture in a semi-formal way those parts of Interlisp which are essential to bring up the remainder of the system. According to the document's abstract,

"In order to implement the Interlisp System (...) on some physical machine, it is only necessary to implement the Interlisp Virtual Machine, since Virtual Machine compatible source code for the rest of the Interlisp System can be obtained from publicly available files."

The VM specifies abstract objects such as "Integers", "Literal Atoms", "List Cells", "Strings", etc., a number of basic LISP functions to manipulate these objects, the control flow and the method of variable binding of the interpreter, and input/output and interrupt processing facilities. Altogether some 250 LISP functions (SUBR's) are defined in the document. Although the VM could be considered as a stand-alone LISP system, the real power of Interlisp lies in the availability of the packages described in the previous section.
2.3 The VM and the rest of Interlisp

It can be seen from the above history that the VM appeared a number of years after most of the Interlisp System was developed. This is probably one of the reasons that the VM specifications are neither completely error-free nor guaranteed to be either necessary or sufficient for implementing the remainder of the Interlisp System. This becomes immediately obvious when one studies the first bootstrap file, PUTDQ, one of the above mentioned "Virtual Machine compatible source code" files. This file is the first file to be read after the VM has been implemented, and it enables the remainder of the Interlisp System to be read in by means of the Interlisp function LOAD. It contains several idiosyncrasies that are illustrative of some of the aforementioned problems. For instance, the file contains the following definition of FIXP:

\[
\text{[LAMBDA (X) } \\
\text{ (AND (NUMBERP X) } \\
\text{ (NOT (FLOATP X))) } \\
\text{ X]}
\]

This will undoubtedly work, because AND, NUMBERP, NOT and FLOATP are SUBR's defined in the VM. But so is FIXP! In a similar manner, IDIFFERENCE is defined as

\[
\text{[LAMBDA (A B) } \\
\text{ (IPLUS A (MINUS B)]}
\]

Presumably the Interlisp-10 Compiler produces good code for this function, but would it perform any better than the SUBR
IDIFFERENCE as defined in the VM? Even worse is the redefinition of the VM SUBR SETPROPLIST:

\[
\text{[LAMBDA (X Y) (CDR (FRPLACD X Y)]}
\]

where FRPLACD is a fast (i.e., no typechecking) non-VM version of the SUBR RPLACD. This definition is particularly offensive because it makes strong assumptions about the underlying representation of both atoms and lists. This kind of information ought to be strictly confined to the VM if the Interlisp System is to be portable.

One of the biggest obstacles to achieving a portable Interlisp System is the creation of an Interlisp Compiler. Because of the sheer volume of Interlisp support software and the amount of processing done by this software during normal Interlisp sessions, no implementation can ever hope to attain production status without the presence of a compiler. Naturally, one would want to implement such a compiler in Interlisp itself in order to keep the Interlisp kernel as small as possible and to improve maintainability. The Interlisp-10 System indeed contains a compiler which produces code for the PDP-10. Many of the aforementioned files, including PUTDQ, rely to some extend on the presence of this particular compiler. Specifically, PUTDQ contains the function LAPRD which not only knows how to read binary files produced by the compiler, but is itself defined in PDP-10 LAP (Lisp Assembly Program)!
One way of determining how closely the VM matches the rest of the Interlisp System is by comparing what each considers to be SUBR's. It turns out that the VM [Moore, 1976] specifies 83 functions (SUBR's) which the Interlisp Reference Manual [Teitelman, 1978] considers EXPR's (and hence these appear throughout the Interlisp System files to be compiled and loaded through bootstrapping); the VM specifies 8 functions that are not recognized as functions in the Manual; and the Manual expects 41 SUBR's to be defined which do not appear in the VM Specification! (For a list of these functions, see Appendix A.)

The author suspects that considerable effort is required to bring up the complete Interlisp System strictly on the basis of the VM. Too many files contain system-, machine-, and implementation-dependent "glitches" such as illustrated above.
3. The Design of the Interlisp Virtual Machine

3.1 Machine Abstraction

The notion of an Abstract Machine as an environment specification has become increasingly popular in the last decade. Two such Machines that have gained wide acceptance are the PCODE Machine for the language Pascal [Nori et al., 1974], and the OCODE Machine for the language BCPL [Richards, 1969]. One key motivation for Abstract Machines is simplicity. The Abstract Machine is defined at a sufficiently low level so that it can be implemented on a real machine with relative ease. As well, the Abstract Machine is specifically designed with a particular high-level language or system in mind so that this language is (presumably) easily mapped into the Abstract Machine.

A second important motivation for Abstract Machines is portability. A software system (e.g., a compiler, text editor, database management system, etc.) may be considered portable if the complete system can be moved to a different machine and/or operating system with little or no modifications to the system required. This is a less restrictive definition than machine independence, since for most useful systems there is some interaction with the environment, and hence the environment interface must almost always be modified for different machines and/or operating systems. We consider a system portable, then, (a) if the system is written in a machine independent language,
(b) the environment interface is clearly isolated and its functionality precisely defined, and (c) no parts of the system make any assumptions about the underlying structure of the environment but rely solely on the specified environment interface. Naturally, there can be some basic specifications which define the scope of portability and on which the entire system may rely (such as byte- and wordsize, presence of a real-time clock, presence of secondary storage, etc.).

LISP has occupied a somewhat unorthodox position in the world of programming languages. Most of the commercial application software relies on standard compiled language environments. As an interpreted system, LISP has a distinctly different flavor. It has been used primarily in research and academic environments for software projects that were hardly ever meant to last beyond a few test runs. In other words, the Artificial Intelligence community, as the principal user of LISP, most often uses it to explore and test new models and theories, without much care for production quality implementations. The software is in a continuous state of flux, and is usually private to a researcher, a small research team, or a department. The need for software sharing beyond these small groups was hardly ever present. Hence, portability has not been such an important issue for LISP'ers as it has been for more commercially oriented software developers. This situation is quickly changing.
Chapter 3

The complexity of AI programs and systems is rapidly increasing. More and more researchers are beginning to feel the need for extensive system support in order to develop, maintain and expand their programs. The need for good LISP compilers is growing as well, in that the increasingly complex programs need faster execution in order to remain usable, or because of real-time constraints. At the same time, the AI community is realizing that the effort of developing extensive system support software, such as Interlisp provides, is quite considerable. The need for sharing is becoming obvious.

Both the AI community and the commercial world are becoming acutely aware of the importance of portability. Not only is there the need for portability across locations (running on different machines and/or different operating systems) so that others may use one's software, which we could term "spatial portability". There is also the need for "temporal portability" -- the ability to transport existing software to new machines as they are developed so that one can hope to use one's own software over the years. The incredible advances in the electronics industry made recently have caused a flood of more and more powerful machines to appear on the market. It is becoming quite obvious that the current crisis in software development is to a large extent due to the fact that so much existing software must be rewritten, or adapted beyond recognition, to accommodate new machines and environments.
As was mentioned above, some fairly successful attempts at portability exist in the form of PCODE for Pascal systems and OCODE for the BCPL language. The clearest example of the viability of this approach is the overwhelming success of UCSD Pascal [Bowles, 1977] which is becoming one of the most popular languages available on microcomputers. A few years ago at the University of California at San Diego, a complete Pascal environment was developed (Pascal to PCODE translator, text editor with knowledge of Pascal source code, symbolic debugger, a file system, and a PCODE interpreter). This environment was originally developed for the DEC PDP-11 but is now available on at least 5 different processors (8080, Z80, 6502, 6800, 9900; soon on Z8000 and 68000). For each new processor, only the PCODE interpreter and some clearly defined I/O and interrupt routines needed to be written or adapted to make the complete system available. The argument against PCODE as being too slow when interpreted was nullified when Western Digital produced the Pascal Microengine, a processor which contained PCODE in microcode. The PERQ workstation from Three Rivers Computer Corporation also runs microcoded PCODE directly at a rate of 1 million PCODE instructions per second [Meyers, 1980].

A similar approach has been made in recent years for LISP systems. At MIT, researchers are working on the development of the Lisp Machine [Weinreb & Moon, 1979] which is a small processor executing LISP primitives and support code in microcode. At Xerox PARC a similar project has been undertaken
Chapter 3

[Deutsch, 1978]. The point of this approach is that once a suitable Abstract Machine has been defined it can be interpreted or emulated on a large variety of machines. So long as the rest of the system only interacts with the Abstract Machine changing the real machine will be completely transparent.

At the same time it has been demonstrated that the Abstract Machine concept has added advantages. Among others, [Deutsch, 1973] has shown that programs can be represented very compactly by the instructions of a suitable Abstract Machine, much more so than by those of any real machine. And Griss and Hearn [1979], who developed a portable compiler for Standard Lisp, have shown adequately that code optimizations can be made quite effectively on the level of the Abstract Machine, or even at the LISP level (e.g., removal of tail recursion, compiling for value or effect, and standard optimization techniques such as removing loop invariants). A third advantage of the Abstract Machine approach is that, since almost the entire system is implemented in LISP, the system is easily maintained, modified or amended using all of the higher level LISP support software.

3.2 The VM Abstraction

The VM goes considerably beyond the Abstract Machine concept as defined above. It is much more an abstraction of the basics of a LISP system than an abstraction of a machine on which a LISP system can easily be realized. Two important indications of support for this view exist. First of all, a
considerable part of the VM Specification is dedicated to the description of Interlisp functions rather than a machine on which these functions are to be implemented. These range from simple, basic functions such as CAR, CDR and CONS, to complex subprograms to perform stack scans and backtraces, string manipulations, file handling, complete I/O, etc. The second indication stems from the fact that the compiler is not a functional part of the VM. The VM places a few restrictions on a compiler if one is present, but there is no indication that any compiler would compile Interlisp code into Virtual Machine code! Since the compiler (or at least an assembler) is not an integral part of the VM, one would assume it is written in Interlisp itself. This would defy the notion of the Virtual Machine as the lowest, basic, self-contained level of the Interlisp System for two reasons:

a. The compiler must assume knowledge of the underlying real machine, as well as intimate knowledge of the particulars of the implementation of the Virtual Machine.

b. The Virtual Machine must be modified to incorporate knowledge of the compiler, and of what it produces, so that the VM functions ARGTYPE, PNTYP, CCODEP, ARGLIST and NARGS can recognize a CEXPR function object, but, more importantly, so that the VM functions EVAL, APPLY, and APPLY* know what to do with it!
3.3 Data Manipulation in the VM

Although not a typed language in the traditional sense, Interlisp provides an ample collection of basic data types, as well as facilities for defining new types. All basic types are defined in the Virtual Machine, and functions are described for the creation, accessing, and modification of instances of these types. Some of these data types are described in the following subsections. The VM function TYPENAME returns the name of the data type of the argument.

3.3.1 List Cells

These objects, with typename LISTP, are the traditional LISP Cons cells. They have the standard two fields Car and Cdr, and are manipulated by the VM functions LISTP, CAR, CDR, CONS, RPLACA, RPLACD and LIST. The VM also maintains a count of List cells created which is accessible and modifiable through the function CONSCOUNT.

3.3.2 Literal Atoms

Literal Atoms, with typename LITATOM, are the traditional LISP identifiers. Recognized by their Pname (print name), they are unique (no two atoms can exist with the same Pname), and eternal once created (i.e. they can not be removed from the system). Literal Atoms have three fields which can be manipulated by the user: a "top level value" field, accessed through the functions GETTOPVAL and SETTOPVAL; a "property list" field, accessible through GETPROPLIST and SETPROPLIST; and a
"function definition" field, accessed through GETD and PUTD.

3.3.3 Numbers

Interlisp recognizes two types of integers: Small Integers, with typename SMALLP, and Large Integers, with typename FIXP. Normally, the representation of integers requires a "box" in which these integers can be deposited. The box can be recognized as such and can be passed around freely without danger of misinterpretation (which would be the case if the integer itself would be passed around). Large Integers (FIXP) are deposited in such boxes. Small Integers, however, are integers of an (unspecified) range which are modified consistently to become addresses into illegal space (e.g., memory occupied by the interpreter itself). For this limited range of addresses no wrong interpretation is possible, so they can be passed around as if they were pointers to objects. Since small integers (those around zero) occur the most frequently, it pays off to modify these to become fake addresses. This is a relatively cheap procedure and does not require the storage allocator (and possibly the garbage collector).

The Virtual Machine requires Character Codes (numeric representations of characters) to be of type SMALLP. (This was presumably done because a large section of I/O implemented in Interlisp relies on Character Code manipulation.) Obviously this is a violation of the virtual nature of the VM since it makes assumptions about the addressing structure of the underlying machine. Omission of the SMALLP type, however, will
create havoc in the higher level Interlisp code, since EQ would fail on two Character Codes representing the same character!

The basic functions for manipulation of Integers are SMALLP, FIXP, FIX, IGREATERP, ILESSP, IPLUS, IDIFFERENCE, IMINUS, ITIMES, IQUOTIENT, and IREMAINDER. In addition the following bit manipulation functions (which take Integers as arguments) are provided: LOGAND, LOGOR, LOGXOR, LLSH (Logical Left Shift), LRSH, LSH (arithmetic Left Shift) and RSH.

Floating point numbers, with typename FLOATP, are always boxed. Functions similar to the Integer ones are provided: FLOATP, FLOAT, FGREATERP, FLESSP, FPLUS, FDIFFERENCE, FMINUS, FTIMES, FQUOTIENT and FREMAINDER. The VM maintains a count of both FIXP and FLOATP boxes, which is accessible through the function BOXCOUNT.

In addition to these type-specific numeric functions, the VM specifies several mixed-mode-arithmetic functions, viz. NUMBERP, MINUSP, GREATERP, LESSP, PLUS, DIFFERENCE, MINUS, TIMES, QUOTIENT, and REMAINDER, as well as the functions EXPT, SQRT, LOG, ANTILOG, SIN, COS, TAN, ARCSIN, ARCCOS, ARCTAN. A (pseudo-) random number generator is also provided (RAND and RANDSET). The function MINUSP is specified in the VM in terms of FMINUSP and IMINUSP, although those are not specified as VM functions!

3.3.4 Strings

Strings in Interlisp have a two level representation. The lowest level is formed by a sequence of characters, somewhere in
memory. Interlisp objects of type STRINGP form the second level. These objects have three fields: a "source" field which points to a sequence of characters; a "position" field containing the index into the character sequence of the starting character of the String; and the "charcount" field, which indicates the number of characters in the String. Two objects of type STRINGP may share the same sequence of characters, hence modifications made to one String may affect other Strings. The "source" field may, presumably for reasons of efficiency, point to the character sequence representing the Pname of a LITATOM. However, in that case a String function which modifies character sequences first creates a copy of the character sequence and deposits a pointer in the "source" field before making the modification. Obviously, any other String still pointing to the Pname character sequence is not affected when this happens. The Virtual Machine specifies the following String manipulation functions: STRINGP, STREQUAL, MKSTRING, CONCAT, RPLSTRING, SUBSTRING, GNC (Get Next Character) and GLC (Get Last Character).

Pattern matching is provided through the functions STRPOS and STRPOSL. The latter function is controlled by a bit table. This is a rather strange entity in the VM — it is an object the functionality of which is clearly defined, which can be created and modified by the function MAKEBITTABLE, and used by the function STRPOSL, and yet the VM does not recognize it as a valid object (i.e., the function BITTABLEP cannot be defined)!
3.3.5 Functional Objects

The flow of control within the VM is governed jointly by the application of functional objects and the behaviour of the run-time stack. Since the application of functional objects is quite traditional we refer the interested reader to [Allen, 1978] for a comprehensive explanation.

The VM recognizes four types of Functional Objects, viz. EXPR's, CEXPR's, SUBR's and FUNARG's. EXPR's are not a type by themselves, but are Lists which are conceptualized as Functional Objects. Further distinction of Functional Objects is made based on the method of their parameter binding, namely whether the value of the arguments (e.g., EXPR's) or the arguments themselves (e.g., FEXPR's) should be bound to the parameters, and whether the Functional Object takes a definite (e.g., EXPR's) or an indefinite number of arguments (e.g., EXPR*). Similarly, the subcategories CEXPR, CEXPR*, CFEXPR, CFEXPR*, SUBR, SUBR*, FSUBR and FSUBR* are recognized. The CEXPR combinations refer to compiled versions of their EXPR counterparts; the SUBR combinations are functions that were hand-coded and assembled by the implementor. A FUNARG object is a list whose first element is the Literal Atom FUNARG, whose second element is a Functional Object, and whose third element is a Stack Pointer which acts as an environment descriptor. FUNARG's will be described in more detail in section 3.4.4.

Interlisp does not require the number of arguments to match the number of formal parameters in the case of spread-type Functional Objects. If more arguments are supplied than can be
bound they are ignored (but evaluated in the case of Eval-type Functional Objects!). If less arguments than parameters are supplied, the remaining parameters are bound to NIL. This provides a flexible and convenient way of supplying defaults to functions.

3.3.6 Other Data Types

The remaining basic data types recognized by the VM are Arrays (ARRAYP), Hash Arrays (HARRAYP), Stack Pointers (STACKP), Read Tables (READTABLEP) and Terminal Tables (TERMTABLEP). Arrays are further distinguished by their contents — either Integers or (pointers to) any Interlisp object. Both kinds of Array have the same type name. The user can distinguish them by means of the function ARRAYTYP. For a description of Hash Arrays we refer the interested reader to Bobrow [1975]. Stack Pointers will be described in section 3.4.3. Read Tables and Terminal Tables are used by the input routines — the former determines the Interlisp syntax (including macros), the latter controls the actions performed on interrupts initiated from the user's terminal.

The VM allows the user to define his own data types with the function DECLAREDATATYPE. This function takes as arguments the name of the new type and a list of field specifications. Valid field specifications are POINTER, FIXP, FLOATP, (SIGNEDBIT j) and (BIT j). The revised edition of the VM [Moore, 1979] omits the (SIGNEDBIT j) as a valid field specification although some functions operating on user defined data types still
recognize it. Instances of user data types can be created and initialized with the function NC\textsc{reate} and their fields may be accessed and modified with the functions \textsc{fetchfield} and \textsc{replacefield}, respectively.

3.4 The Run-time Stack

3.4.1 A Model of Multiple Environments

The Interlisp run-time stack structure is based on a model of multiple environments proposed by D.G. Bobrow and B. Wegbreit [1973]. This model was defined in recognition of the fact that for many control and access environment structures the traditional last-in first-out stack discipline is inadequate. Many recent programming techniques such as backtracking, multiprogramming and the use of coroutines require that stack storage space associated with function activations be retainable independent of the flow of control. The Bobrow and Wegbreit model allows for arbitrary retention of activation records, yet behaves exactly like a LIFO stack if retention is not required.

For each function activation two storage components are distinguished, viz. data storage and control storage. Data storage is reserved for function parameters and named variables local to the function. This storage is fixed in size since these storage requirements do not change during the execution of the function body. Control storage is used for temporary intermediate results of computation plus any other relevant information pertaining to the function activation and hence
varies in size during the execution of the function body.

In the Bobrow and Wegbreit model data storage is contained in the "basic frame" and control storage in the "frame extension". The frame extension contains three pointers to other frames. The BLINK is a pointer to the associated basic frame. The ALINK is a pointer to a chain of frame extensions constituting the access environment. This chain is used for accessing the values of free variables (i.e., all variables that are not bound in the function module's own basic frame). The CLINK is a pointer to a chain of frame extensions constituting the control environment. When execution of the function body terminates control is passed back to the function whose frame extension is the first in the control chain. This method of generalized return is made possible through the existence of the "Continuation Point" field in the frame extension. At the time of function invocation the caller rather than the callee saves the continuation point of the computation in his own frame extension. This allows the caller to be reactivated independent of whether control is returning from the callee or from some other frame.

Additional fields given for frame extensions are the "Exitfn" field specifying actions to be taken before the function exits; the "Framename" containing the name of the function; the "ReturnType" field for type checking of returned values when the function body is reactivated; the "USE" field containing a reference count of the frame extension; the "Size" and "Max" fields for memory management; and an unspecified
number of "Temporary" fields. The basic frame also has "Size" and "Max" fields as well as the "CXT" field containing a count of the number of frame extensions pointing to the basic frame. Bobrow and Wegbreit specify the primitive functions Getexfn and Framenm to obtain the values of the "Exitfn" and "Framename" fields, respectively, and the function Setexfn to alter the value of the "Exitfn" field.

In a normal function call and return sequence the basic frame of the callee will have an extension count of one and both the ALINK and the CLINK of the frame extension will point to the caller's frame extension. When the function exits both the basic frame and the frame extension are deleted and control returns to the caller.

Retention of stack frames is made possible by the introduction of a special data type called "environment descriptor" (ED). Bobrow and Wegbreit define the four primitive functions Environ, Setenv, Mkframe and Enveval which create and alter ED's, create new frames with specified context links, and allow execution of a computation in a specified context. Whenever control returns to a function whose frame is referenced from other frames or from one or more ED's a copy of the frame is made and the computation is continued in the copy. When the function exits, its frame (i.e., the copy) is deleted but the original frame is retained.

Several ways are provided for the user to specify frames:

1. An integer N: (a) N = 0 specifies the frame which called the primitive function using the frame specification; (b)
N > 0 specifies the frame N links down the control link chain from the N = 0 frame; and (c) N < 0 specifies the frame |N| links down the access link chain from the N = 0 frame.

2. A list of two elements (F, N) where F is a framename and N is an integer. This gives the Nth frame with name F, where the sign of N specifies which environment chain to descent.

3. The constant NIL. For a frame with NIL in its ALINK field global values of free variables are to be used. If control returns from a frame whose CLINK is NIL then the system halts. If an ED is set to NIL then the storage it previously referenced is released.

4. An environment descriptor. The frame specified is the frame referenced by the ED.

5. A list of one element which is an ED. The frame specified is the frame referenced by the ED. The primitive function using the frame specification will execute Setenv(ED,NIL). This is sometimes necessary in the case of Enveval since the user cannot do this explicitly before the call to Enveval because this would invalidate the ED, nor can he do this after the call since control may never return to that point.

3.4.2 The Spaghetti Stack

The run-time stack structure in the VM is an adaptation of the Bobrow and Wegbreit model as described in the previous section. Several differences exist that make the spaghetti
stack behave slightly differently. The VM explicitly warns that some differences exist, but does not attempt to enumerate these. In the following paragraphs we shall discuss these differences in detail.

In the VM, the "framename" field is contained in the basic frame rather than in the frame extension. Because frame extensions can occur more frequently than basic frames (basic frames can be shared) this leads to some (rather insignificant) savings in space, but it requires an extra pointer dereference to obtain the framename (through the BLINK field of the frame extension). There is no difference in functionality.

The VM Specification of the spaghetti stack makes no mention of the "Size", "Max", "USE" and "CXT" fields. The method of stackspace allocation and deallocation is left entirely to the implementor.

The Interlisp spaghetti stack model has no provisions for exit functions. Hence the "Exitfn" field and the functions Getexfn and Setexfn specified in the Bobrow and Wegbreit model do not exist.

The VM recognizes the need for a "Continuation Point" field but does not explicitly incorporate this in the frame extension nor is it referred to in any other part of the VM Specification. Instead, the field is considered part of the "Temporaries" and its maintenance is left to the implementor.

Interlisp provides greater control over the contents of frames than the Bobrow and Wegbreit model allows:

a. The functions STKNAME and STKNTHNAME allow one to obtain the
contents of the "framename" field and the function SETSTKNAME modifies this field. SETSTKNAME is a SUBR in the Interlisp Reference Manual but is omitted from the VM Specification.

b. The functions STKNARGS, STKARG, STKARGNAME, SETSTKARG, SETSTKARGNAME, STKSCAN and FRAMESCAN allow one to inspect or modify the bindings in a basic frame.

c. The functions SETALINK and SETCLINK modify the corresponding fields in a frame extension. They test for circularity so that the integrity of stack chains is maintained. Allowing a frame to be a member of its own ALINK or CLINK chain would cause an infinite loop in passing control down the CLINK chain as well as in free variable lookup down the ALINK chain. These functions are VM (revised edition) SUBR's but do not appear in the Reference Manual.

d. Frames and frame sequences can be created with the VM SUBR's MKFRAME and COPYSTK. MKFRAME does not appear in the Reference Manual.

3.4.3 Stack Pointers

The VM spaghetti stack equivalent of Environment Descriptors (ED's) are Stack Pointers. Stack Pointers can be created by means of the functions STKPOS, STKNTH, MKFRAME, COPYSTK, STKSCAN and FUNCTION, recognized as such with the predicate STACKP, and tested for equality with the function EQP. A copy of a Stack Pointer, SP, can be obtained by executing (COPYALL SP) or (STKNTH 0 SP). The latter method is given in the Reference Manual but will not work according to the VM
specification of the function STKNTH (presumably the VM definition is erroneous).

The spaghetti stack model allows one to specify frames in a variety of ways as well but the differences are noteworthy:

1. An integer \( N \): (a) \( N = 0 \) specifies the frame which belongs to the stack function itself, not the frame which called the primitive function using the frame specification; (b) \( N > 0 \) specifies the frame \( N \) links down the access link chain from the \( N = 0 \) frame; and (c) \( N < 0 \) specifies the frame \( |N| \) links down the control link chain from the \( N = 0 \) frame. Hence the meaning of the sign is reversed from the Bobrow and Wegbreit model.

2. The constant NIL. This refers to the frame of the currently active function (i.e., the stack function itself).

3. The constant T. This refers to the frame of the top level process (i.e., the process executing EVALQT).

4. Any atom other than NIL or T. This specifies the first frame down the control link of the active frame (the frame of the stack function) whose framename is the atom given.

5. A Stack Pointer. The frame specified is the frame referenced by the Stack Pointer.

Because the constant NIL has different semantics from the Bobrow and Wegbreit model a different way of releasing and reusing Stack Pointers has been defined. The functions RELSTK and CLEARSTK are provided to explicitly release stackspace referred to by a Stack Pointer or by all Stack Pointers, respectively.
Implicit release is provided for by means of additional arguments to the stack functions described above. Flag arguments specify releasing a Stack pointer after the referenced frame has been obtained, and optional Stack Pointer arguments allow the VM to reuse these Stack Pointers. The predicate RELSTKP can be used to determine whether a Stack Pointer is referring to a stack frame or has been released.

3.4.4 Altering the Flow of Control

The VM defines a number of functions which take frame specifications as arguments and serve to alter the flow of control or allow computations to be performed in a given environment. The simplest of these are the functions RETTO and RETFROM which cause control to return to (from) the specified frame. EVALV obtains the binding of the first argument (which must be a Literal Atom) in the access environment (the ALINK chain) specified by its second argument. ENVEVAL and ENVAPPLY create a dummy basic frame with framename NIL and no bindings, and a dummy frame extension with ALINK and CLINK as specified by their arguments, after which the form is evaluated, or the function applied, using this dummy frame as the starting frame of the current access and control environment.

FUNARG Functional Objects provide a general method of performing a computation in the current control environment using a different access environment. FUNARG's are created with the Noeval-type function FUNCTION which takes two arguments: a Form and an Environment Specification (ES). If ES is NIL then
FUNCTION simply returns Form, so in this mode FUNCTION is equivalent to QUOTE, except that it signals to the compiler that Form is a Functional Object and should be compiled accordingly. This is useful, for instance, for compiling open Lambda expressions. If ES is not NIL then it can be either a Stack Pointer or a proper list of Literal Atoms. (According to the Interlisp Reference Manual, ES may also be a Literal Atom, in which case it is evaluated first; this is not specified for the VM.) If ES is a list a basic frame is constructed with framename NIL and with bindings made up of the atoms in the list and their values in the current access environment. A dummy frame extension is created for this basic frame with ALINK equal to the current access chain and CLINK set to NIL, and FUNCTION returns a list of three elements: the Literal Atom FUNARG, Form, and a Stack Pointer to the frame just created. If ES is a Stack Pointer then FUNCTION just returns a list of the form (FUNARG Form ES).

When a FUNARG expression is applied to an argument list, a dummy frame is created with its ALINK set to the frame referenced by the Stack pointer (third element in the FUNARG expression) and its CLINK set to the current control chain. Then the Form of the FUNARG expression is applied to the original argument list in this environment.

3.4.5 Blip Fields

Interlisp provides a means to access directly some fields on the stack which pertain to the flow of control. These
fields, called "Blip fields" are associated with the functions EVAL, APPLY, COND, PROG, PROGN and PROG1. The last four use the blip fields *FORM* and *TAIL* to hold the current form being evaluated and the list of remaining forms to be evaluated, respectively. EVAL and APPLY use these as well as the blip field *FN* and a number of blip fields *ARGVAL*. The field *FN* holds the name of a function to be called, and the fields *ARGVAL* hold intermediate values of (evaluated) arguments to the function. The VM leaves it to the implementor to decide whether these blip fields are incorporated in the "Temporaries" field of the frame extension or are bindings in the basic frame of the blip-using functions. The blip fields can be accessed and modified with the functions BLIPSCAN, BLIPVAL and SETBLIPVAL. Access to these fields enables the Interlisp error processor (particularly, the DWIM facility) to inspect (and possibly alter) the flow of control and to correct erroneous forms on the stack.

3.4.6 Copying Frames

Because the VM does not explicitly state all of the differences between the spaghetti stack and the Bobrow and Wegbreit model, a glaring omission in the VM Specification is easily assumed by the implementor to be an intended deviation from the original model. The VM, although very precise and specific in most cases, fails to specify that frames should be copied whenever control returns to them and their reference counts are greater than one. This copying is also not mentioned
in the Stack section of the Interlisp Reference Manual [Teitelman, 1978]. It is a rather important point because failing to implement this causes functions using the spaghetti stack facilities to behave quite differently. Consider the following definition of the function FOO:

```
(NLAMBDA (STP)
  (PRINT 'Hi)
  (LAMBDA (FRAME)
    (COND ((STACKP FRAME) (SET STP FRAME))
      (T (PRINT FRAME)
        (STKPOS 'FOO))
        (PRINT 'There)
        'FOO–exit)
```

The arguments to STKPOS are "framenname", "nth such frame", "starting frame" and "reusable stackpointer". Hence, (STKPOS 'FOO) will search for the first frame down the control chain whose frame name is FOO, starting the search with the currently active frame and returning a new Stack Pointer to this frame if found.

When frames are copied as described above, the following dialogue would ensue:

```
* (FOO BAR)
>   Hi
>   there
>   FOO–exit
* (RETTO BAR 'Hello)
>   Hello
>   there
>   FOO–exit
*
```

The function RETTO takes three arguments, viz. "frame to return to", "value returned" and "release frame flag". (For a detailed
execution trace of this example see Appendix F.) If frames are not copied, however, then all forms within FOO after the call to STKPOS will be executed within the original FOO frame, and returning to it will have no effect:

* (FOO BAR)
  > Hi
  > there
  > FOO-exit
* (RETTO BAR 'Hello)
  > Hello
*

In personal communications with researchers at Xerox PARC [Masinter et al., 1980] the author was informed that the spaghetti stack conforms to the Bobrow and Wegbreit model in this respect and hence that frames should be copied when reactivated with a reference count greater than one (the one accounting for the implicit Stack Pointer *ACTFRAME* which always points to the currently active frame).
4. The Implementation of the Virtual Machine

Multilisp is a (partial) implementation of the Interlisp Virtual Machine developed at the University of British Columbia. The system is written in Pascal [Wirth, 1971] and consists of approximately 16500 lines of Pascal source distributed over 21 files. Each file contains a relatively self-contained section of the system, such as memory management, input, output, interpreter, and logical groups of VM functions. Multilisp was designed to be portable and is currently running on both the Amdahl 470 under MTS and the DEC VAX 11/780 under VMS. Developed on the Amdahl in approximately 12 man-months, about 3 man-weeks were required to transport Multilisp to the VAX.

Multilisp is intended primarily for large-scale systems with an autonomous disk operating system supporting virtual memory. Although developed for a machine with a wordsize of four bytes (32 bits), the system does not rely on this fact. The only assumptions made are that memory is byte-addressable, that a byte can contain a Character Code, and that a word is sufficient to hold any memory address.

In this Chapter we shall describe the Multilisp system briefly and discuss some of the design issues that were encountered in the implementation of the VM.
4.1 The Representation of Objects

Although not all Interlisp meta-objects have a corresponding Object associated with them in the VM Specification (e.g., Bittables) they need to have some recognizable representation. Hence Multilisp introduces at least one (internal) type name for every Object. The layout of Objects is specified with a Pascal record with case variants based on the object type (see Appendix B). All Interlisp objects have been implemented in Multilisp with the exception of Terminal Tables for reasons which will be given in Chapter 5. In the following subsections we shall discuss the interesting objects in more detail.

4.1.1 List Cells

A large number of researchers have studied the efficient representation of LISP Cons cells ([Hansen, 1969], [Cheney, 1970], [Wegbreit, 1972], [Clark & Green, 1977], and [Morris, 1978], among others). Clark and Green found in their extensive study of list structures that the CDR fields of almost all list cells (>98%) point themselves to other list cells or to NIL. As well, they found that list cells are generally linear and pointed to by only one other list cell, so that the CDR of a list cell is frequently allocated very near to the list cell itself. Rather than allocating a full word for the CDR field of a list cell which can hold any full memory address one can make use of this behaviour of lists by encoding the distance of the CDR from the list cell in the list cell itself. Since this
distance is generally small this encoding, referred to as CDR-Coding, requires very few bits. Because in approximately one out of every four cases [Clark & Green, 1977] the CDR is NIL, encoding this information as well further reduces memory requirements. This form of list compaction can save up to half the total memory requirements of ordinary list cells. In paged memory systems this has the added advantage of producing fewer direct memory references, and hence fewer page faults.

In Multilisp we have adopted the scheme of Immediate CDR-Coding. This scheme recognizes four different types of list cells:

a. A list cell whose CDR is NIL
b. A list cell whose CDR is the cell immediately following
c. A full list cell (with both CAR and CDR fields)
d. An indirect list cell

Indirect list cells are cells that were at one time Compact List cells, but whose CDR was replaced by something other than NIL. A Full List cell had to be constructed to hold the original CAR field value and the new CDR field value. The address of the Full List cell was left in the original Compact cell which was marked as an Indirect cell.

4.1.2 Literal Atoms

Literal Atoms are hashed into a hash table by means of a polynomial hash function on the Pname, using linked lists for conflict resolution. In order to avoid the use of LISP lists in hash buckets, Literal Atoms have a fourth field, the "Hash
Link", which is a pointer to the next atom in the hash bucket. The advantage is, of course, that CONS need not be called for insertion of a new Literal Atom, CAR and CDR need not be called for searching through the hash bucket, and the Garbage Collector need not to trace the hash buckets. Otherwise the fields are as described in Chapter 3.

4.1.3 Numbers

The implementation of boxed numbers (i.e., FIXP and FLOATP) is straightforward (see Appendix B). Meeting the VM requirement of unique representation of at least all Character Codes, however, proved to be somewhat more difficult. Under a large time-sharing system such as MTS or UNIX it cannot be assumed that the implementor is able to obtain and use the address range of memory where the system loader has loaded the object program constituting the VM. Hence, using this address range for SMALLP numbers is not feasible. Moreover, Multilisp requires all objects to be referenced in order to obtain their object type. (This will be explained in the next section). Multilisp, therefore, acquires at system start-up time a section of memory from the operating system which is designated as SMALLP space. This space remains the same throughout the time Multilisp is running. At each valid address within this space the type indicator SMALLP is deposited once at the time of initialization. All Small Integers are mapped into this space by adding the address of the middle of the SMALLP space to the value of the Small Integer, giving all Small Integers (between
-1024 and 1023 in the current implementation) a unique representation. Each address so obtained, when dereferenced, yields the object type SMALLP, yet for all Small Integers neither the heap space allocation mechanism nor the garbage collector need be invoked.

**4.2 Object Memory Management**

An important issue in the design and implementation of large-scale LISP systems is the way object memory is organized. Nowhere is the time-space dilemma more apparent than in the object memory manager. In general, the more emphasis is placed on efficient memory allocation, the more processor time is required to realize it. CDR-Coding, as it was described in section 4.1.1, is an obvious example. This scheme can save up to 50% of List cell memory space, but the price one pays is encoding and decoding of the List cells themselves. In the case of List cells the memory savings are sufficiently significant to justify the increased complexity, but in other cases this is not immediately obvious.

Two important criteria have influenced the design of the object memory in Multilisp. First of all, the requirement of portability of Multilisp did not allow for using memory words containing pointers for any other information. For instance, the Amdahl 470 (an IBM 370 derivative) uses a wordsize of 32 bits, but only supports a 24 bit address space, and one is tempted to use the high order byte for other information. On
the other hand, although the DEC VAX 11/780 also uses a wordsize of 32 bits, its address space is a full 32 bits as well. Secondly, on the particular system Multilisp was developed (the Michigan Terminal System, supporting large-scale time-sharing with accounting) memory is relatively cheap to acquire but processor time is a precious commodity. Hence, except for list compaction, in the time-space tradeoff decisions savings in processor time were usually favoured over space savings.

4.2.1 Object Typing

There are basically three methods for establishing the type of objects in object memory:

a. The Boundary Check Method Objects are allocated in fixed segments of memory, each segment containing objects of only one type. The type of an object is then determined by sequentially comparing the address of the object with the known segment boundaries until the right segment is found. This method is generally unsatisfactory because type determination requires search, and because expanding segments to allow for more objects of a certain type is usually not feasible. The method can be appropriate where the implementor has complete control over the entire address space and can allocate segments at absolute addresses.

b. The Pagetable Method This is a far more flexible method than Boundary Checking. It is applicable both in environments where full control of the entire address space is available and in
those environments where a (possibly uncooperative) operating system is responsible for page allocation. Each page is dedicated to one type of object only. Pages are allocated preferably on demand, and once allocated the entry in the pagetable corresponding to the page number is assigned a type indicator. Whenever more memory is required for a certain object type a new page is requested from the page manager, and its associated pagetable entry updated. Object types are then determined by the code sequence

```
ObjectType := PageTable[ObjectAddress DIV Pagesize];
```

With this method space is hardly ever wasted (i.e., allocated for a certain type but not currently in use). As well, memory partitioning on the basis of types is dynamic and dependent only on the characteristics of the total program configuration and the run-time space requirements. The only memory "wasted" is in the form of gaps left at the end of a page. Since type allocation is per page, where consecutive pages allocated for a particular type are not necessarily contiguous, larger objects such as arrays may not be able to use the remainder of the current page but must be allocated in a new page, leaving a gap in the current page.

c. The Tagged Object Method With this method all objects are allocated from one "heap". Each object has a "type indicator" field, and hence the type of an object is determined by the code sequence
ObjectType := Object@.TypeField;

The basic attraction of this method is simplicity -- both in object allocation and in type determination. Since there is only one heap only one pointer is required to indicate the current heap top, whereas the Pagetable Method requires a top-of-page pointer for every type. It is true that each reference with the Tagged Object Method could cause a pagefault, but this is true also for indexed accessing of the pagetable, unless the page manager can be instructed to keep the entire pagetable permanently in the active working set (the set of virtual pages mapped onto real memory). It is also true that the Tagged Object Method is potentially wasteful of memory, especially in systems where word alignment is enforced. On the other hand, any garbage collection method will require one or more bits for marking, and if no assumptions can be made about the number of address bits in a pointer, extra space is required anyway. If extra bits are available in a pointer word then these can be used for tags. However, in that case a price is paid in extra processing since every pointer dereference will then require masking out the tag bits. As well, with this method CDR-Coding comes almost for free by recognizing four different types of Listcells instead of one.

Because of the simplicity of memory management with the Tagged Object Method (and hence a simpler garbage collector as well), and the fast method of type determination this last method was chosen for the Multilisp system. However, the
Pagetable Method is recognized as a serious contender. Unfortunately, resources were not available to experiment with both methods and to choose on the basis of performance statistics. Instead, the choice was made purely on reasoning and intuitive appeal. (For a summary of the fixed memory requirements of objects in Multilisp for both the Amdahl 470 and the VAX 11/780 see Appendix B. The differences are due to word alignment and packing of Pascal records, or the lack thereof).

List cells are allocated from the top of the heapspace so that CONS is able to use compact representations when a list is being constructed. Atoms are allocated in a separate space so as to avoid unnecessary garbage collection (atoms can be created but not deleted). Character sequences also are allocated in a separate space recognizing the need for a special, string-specific garbage collector (not yet implemented). All other objects are allocated at the bottom of the heapspace. When the top and bottom pointers of the heapspace meet the garbage collector is invoked.

4.2.2 Garbage Collection

A significant number of papers have been published on the subject of garbage collection. For a system which supports CDR-Coding of List cells, the compactifying garbage collector is the natural choice ([Hansen, 1969], [Cheney, 1970], [Wegbreit, 1972], [Deutsch & Bobrow, 1976], [Morris, 1978], [Baker, 1978]). Multilisp, with its single heapspace, has a non-recursive, copying garbage collector based on Cheney's Algorithm. Garbage
collection is initiated by acquiring a second heapspace from the operating system, its size at least equal to the size of the current heapspace (the actual size is a function of the amount of active heapspace after completion of the previous garbage collection). Then a trace is started from the atom hashtable, the run-time stack and all system pointer variables. Each object thus encountered is copied to the bottom of the new heapspace. If a List cell is encountered then the entire list (in the CDR direction) is copied, jumping through possible Indirect cells, and making all copied cells compact if possible (see Appendix C). A bottom marker moves along the new heapspace as objects are added. Once an object has been copied to the new heapspace the old object is smashed. Its "type indicator" field is replaced by a special "type" FORWARD, and the address of the copy is left in the original object. (This of course assumes that every object is at least large enough to contain both a forwarding address and the special type mark FORWARD). When all directly accessible objects have thus been traced and copied, a trace of the new heapspace is initiated by having a scanning pointer, starting at the bottom of the new heapspace, repeat the process just described for every relevant pointer contained in copied objects, until the scanning pointer catches up to the bottom pointer. At that point the garbage collection is complete, the old heapspace can be discarded, and the size of the next heapspace determined on the basis of how much of the new heapspace is active.

A special push-down stack is employed for maintaining
temporary pointers into the heap. Whenever some part of Multilisp invokes the storage allocator, and thereby potentially causes a garbage collection to occur, relevant local pointers are pushed on this special stack, and popped again on return from the storage allocator. The garbage collector checks this stack during the initial trace of immediately accessible pointers.

4.3 The Spaghetti Stack

Because of its complexity the implementation of the spaghetti stack is a non-trivial task. A number of observations have influenced the design of the current implementation:

a. A fairly large amount of storage is required. Firstly, each function invocation consumes more space than is the case in more conventional stack environments, because the spaghetti stack has more overhead: neither the ALINK nor the CLINK of a frame necessarily point to the frame immediately above it, hence an additional pointer is required pointing to the frame immediately preceding the current frame so that storage occupied by the latter can be returned to the former; as well, frames below the current one may be retained so that the current frame must be aware of its local "ceiling", which may not be equal to the end of the stack. This stack maintenance overhead is aggrevated because it is necessary for both the basic frame and the frame extension since they can exist independent of one another. Secondly, if frames are retained than all their ancestor frames are necessarily
retained as well, which can render large sections of the stack unusable.

b. The stack manager must be able to make efficient use of "holes" -- these occur, for instance, when three branches have been allocated consecutively and the middle one is subsequently released.

c. The integrity of the stack must at all times be preserved. This is especially critical in the case of stack overflow.

d. Under operating systems such as MTS, where computing cost and performance depends on, among other things, the amount of virtual memory acquired from the operating system, it is desirable to return unused stack space to the system as soon as possible.

Maintenance of the spaghetti stack in Multilisp occurs on three levels: the lowest level is concerned with unstructured stack space called segments; the next level views the stack as a collection of stack "elements"; the third level deals with frames as described in Chapter 3.

4.3.1 Stack Segments

Segments are blocks of contiguous memory acquired from the operating system. Segments are not necessarily equal in size nor are they contiguous. Initially, one segment is allocated which is pointed to by CURRENT_SEGMENT. Segments are maintained with a separate static "expansion stack" allowing for a fixed (but easily changed) number of segments. Each entry in the expansion stack has three fields, viz. SEGMENT_START,
SEGMENT_END and SEGMENT_TOP. The SEGMENT_TOP field marks the
top-of-stack local to the segment. When the CURRENT_SEGMENT is
full and more stack space is requested by the next higher level,
the Segment Manager scans all other existing segments for one
which has a SEGMENT_TOP less than SEGMENT_END (and therefore has
room to spare). If such a segment is found it is made the
CURRENT_SEGMENT; otherwise a new segment is acquired from the
operating system, made to be the CURRENT_SEGMENT, and pushed
onto the expansion stack. If this fills up the expansion stack
a Stack Overflow error is signaled. (Hence, the Interlisp error
processor can run in this last segment). If this segment
overflows as well, nothing can be done and Multilisp aborts.
Whenever a segment becomes empty (signaled by the next higher
level) the Segment Manager scans the expansion stack for a
second segment marked as empty. If it finds one it is returned
to the operating system and the expansion stack is popped
accordingly. This method of releasing the second segment
provides the necessary hysteresis, which if not present would
cause tremendous thrashing in boundary cases. For instance, if
the body of a PROGN is executing at the end of a segment then
for the execution of each form inside the PROGN a new segment
would repeatedly be acquired from the operating system and
released again if hysteresis were not provided.

4.3.2 Stack Elements

Stack elements are created at the top of the
CURRENT_SEGMENT. They form a doubly linked list with two
fields, LAST and NEXT, per element. LAST points to the element immediately preceding the new element and NEXT points to the element immediately succeeding the new one (at the time of creation, this is equivalent to the end of the segment). Each element has a third field, TOP, which points to the top of the current element. Hence, the difference between the address of an element and the address contained in its TOP field corresponds to the contents of the "Size" field in the Bobrow and Wegbreit model, and the difference between the NEXT field and the TOP field of an element corresponds to the "Max" field in the model. Actual addresses were used instead of these "Size" and "Max" fields in order to avoid arithmetic and thus to speed up the Element Manager.

Releasing an element is done with the standard algorithm for deletion of elements in a doubly-linked list [Knuth, 1968]. When new segments are acquired a dummy element is created at the bottom of the segment whose LAST pointer contains NIL, and whose NEXT pointer is the same as the SEGMENT_END. Hence, if the space occupied by an element is returned to an element which has the aforementioned LAST and NEXT characteristics the segment must be empty and this is signaled to the Segment Manager.

4.3.3 Stack Frames

Basic frames and frame extensions are created (by overlay) within elements (see Appendix D). The function CRE_BASIC allocates basic frames of fixed size, enough to hold the fixed contents NAME, NARGS and REFCNT, and the number of bindings
given as argument. The TOP field of the element containing the basic frame points just past the last binding. Frame extensions are allocated by MAKE_XFRAME in an element big enough to hold the fixed contents, such as USECNT, ALINK and CLINK, and some stack blocks (described below). The TOP field of the element initially points just past the last fixed member of the frame extension.

Multilisp controls the stack via the frame creation and deletion functions and accesses the stack through the internal stack pointer *ACTFRAME* and all dynamically allocated Stack Pointers, and hence has access to all frame extensions. Basic frames are only accessible through their corresponding frame extensions, and every frame extension must have a basic frame. This poses a problem in case of stack overflow: either a basic frame is created first and the stack overflows when attempting to create the matching frame extension, in which case there is a "ghost" element on the stack that is not pointed to by any frame extension, yet has been linked on the element level, which could tie up an entire segment; or a frame extension is created first and the stack overflows when attempting to create the corresponding basic frame, in which case there is a frame extension that has been linked with the rest of the stack (i.e., ALINK and CLINK, not LAST and NEXT) but that has no basic frame, violating the integrity of the spaghetti stack. MULILISP's solution is to consistently create basic frames first, then the frame extension. CRE_BASIC sets the global pointer CURRENT_BASIC to the newly created basic frame; MAKE_XFRAME sets
it to NIL. When the internal error routine is called (which in turn will invoke the Interlisp error handler) it checks if CURRENT_BASIC is NIL, and if not it deletes this "ghost" frame.

4.3.4 Temporaries and Blip Fields

As can be seen from the above sections, creating and deleting of basic frames and frame extensions is an elaborate process. In order to reduce stack activity internal calls to EVAL do not cause creation of the EVAL frame sequence. Instead, an eval-block is pushed onto the frame extension that is calling EVAL (see Appendix D). These blocks are allocated at the TOP of the element containing the frame extension after which the TOP pointer is adjusted. If there is no more room to expand the frame extension, the frame is copied to a location where expansion is possible, and the old frame deleted. Deletion of the old frame is safe because the function requiring this expansion is currently active. If there were any Stack Pointers referencing the function's frame then it is now running in an (unshared) copy of itself. The eval-blocks provide room for the blip fields and the continuation point.

A slightly different block is used for holding temporaries. This type of block is used by the function READ, which requires a certain number of temporaries for several purposes very frequently.

A possible alternative to the scheme of eval-blocks is to consider the dynamic portion of the frame extension as a varying size array of words, where a word may contain a heap pointer, a
stack pointer, an integer, a continuation point, or any other value. This method could result in some memory savings and therefore reduce the chance of having to copy the frame extension for lack of expansion space. However, besides the difficulty referring to such words in Pascal, this approach requires associating indicators with every word indicating what the word is used for, so that the various functions dealing with the stack (including the garbage collector) know how to interpret the words. The indicator would occupy 1 byte on the VAX-11, but on the Amdahl 470 it would take another word so that alignment is guaranteed. It was decided that the efficiency of this approach is too dependent on the underlying machine. As well, the eval-block approach assures fast access of the fields inside blocks (i.e., we know the offsets of all fields such as *FORM* and don't need to scan the stack in order to find them). However, a compiler may necessitate this approach in order to make efficient use of the stack.

In addition to the internal stack pointer *ACTFRAME* Multilisp has access to the currently active block through the internal stack pointer *ACTBLOCK*. Blocks are created and deleted by CRE_BLOCK and REL_ACTBLOCK which maintain the value of *ACTBLOCK* as well, and always apply to the currently active frame. Thus, *ACTBLOCK* always points to the topmost block of *ACTFRAME*. A frame extension may have more than one block because of EVAL's recursion, e.g. in order to evaluate:

(CAR (FOO (BAR ...)))

the interpreter needs a block to evaluate the argument to CAR.
This evaluation, in turn, needs a block to evaluate the arguments to FOO, etc.

4.3.5 Argument Binding

Whether to use deep binding or shallow binding in LISP systems has been a controversial topic for several years. The general argument is that variable access is a more frequent operation than variable binding and hence employing a shallow binding technique should result in better system performance. This is true only for programs which consist of many large functions and PROG's. For programs that use small function definitions and many open LAMBDA's variable binding obviously becomes more crucial than variable access. Another factor in determining which binding method is more advantageous is whether user programs primarily follow the traditional flow of control or make use of multi-programming, coroutines and backtracking. As soon as environment switching becomes a significant part of program execution, overall system performance quickly deteriorates if shallow binding is employed.

The VM Specification for variable binding follows the deep binding paradigm. Of course, implementing either shallow or deep binding is completely transparent to the user (other than by its effect on performance) so the implementor can deviate from the VM Specification. Multilisp has opted for the deep binding regime. The user community for which it was originally developed makes heavy use of the spaghetti stack facilities for environment switching and the cost for this in a shallow-bound
system can be high. Also, deep binding in the context of the spaghetti stack is considerably easier to conceptualize and to implement than shallow binding. Unfortunately, resources were not available to obtain substantial performance statistics in order to make a decision on the basis of these. If shallow binding were to be implemented the best scheme for environment switching would probably be Baker's rerouting algorithm [Baker, 1978].

4.4 The Multilisp Interpreter

4.4.1 The Basic Loop

On system startup the stack is initialized with a basic frame with framename T and no bindings, a frame extension with NIL in both the ALINK and CLINK fields, and an eval-block with a routine executing EVALQT as continuation point. This frame sequence is in the VM referred to as the "top frame". Control can return to this frame but can never return from it (attempting to return from a frame which has NIL in its CLINK causes an error). The interpreter is continuously reactivated by obtaining the continuation point from *ACTBLOCK* and calling the corresponding continuation routine. The continuation point is an index into a branch table rather than an actual machine code address (see Appendix E). Since Multilisp is implemented in a high level language it is not only impossible to obtain the address of any actual machine instruction but also completely undermines the aim for portability of Multilisp. In addition,
using indices allows one to modify the actions taken for continuation of the system without affecting any other part of the system. It turns out that very few continuation points (24) need actually be distinguished because for intermediate calls and returns of all the Multilisp support routines the Pascal stack is used rather than the spaghetti stack. Before the continuation procedure for the active frame is invoked a check is made of the frame's use count. If it is greater than one the active frame is split so that processing can continue in a unique copy of the frame.

Precisely because the Pascal stack is used for internal calls there exists the danger of running out of Pascal stack space. This, of course, will happen if the internal call to EVAL is recursive on the Pascal stack as well as the spaghetti stack. Unwinding the Pascal stack independent of the flow of Multilisp control is done by placing the continuous interpreter loop in the main body of the program, preceeded by a (global) label (see Appendix E). Multilisp assumes the convention that whenever the internal routines EVAL, APPLY, ERROR or POP_ACTFRAME are called a branch is executed to this global label, thus popping the Pascal stack. Hence, whenever an internal routine calls these routines and control is expected to return for further processing the return will not be made according to the Pascal regime. Instead, a new block must be provided holding the index of the proper continuation routine.
4.4.2 Argument Evaluation

When EVAL encounters a list whose first argument is a Functional Object of Eval-type all arguments need to be evaluated before the function can be called. For this purpose an eval-block is created on top of the currently active frame. The *FN* blip field is set to the Functional Object, the *TAIL* field is set to the list of arguments, and the continuation field contains the index for CNT_EVLIS, the continuation procedure dealing with argument evaluation. Each time this procedure is entered (through a call from the main interpreter loop) the returned value is pushed onto the currently active eval-block as a new *ARGVAL* and EVAL is called again on the next argument by CDR'ing down the *TAIL* field, until the argument list is exhausted. At this point, after checking that the *FN* blip field is still a valid Functional Object (it could have been changed by SETBLIPVAL in a lower function call), the function can be called. This means that the formal parameters of the function must be bound to the values of the arguments contained in the *ARGVAL* fields. Ideally, the *FN* field and the *ARGVAL* fields are placed in the eval-block in such a way that the eval-block can be overlayed with a new basic frame. The framename and the bindings would then be in place and only the formal parameter names would have to be filled in, thus avoiding some unnecessary copying. This, however, would require that we know precisely how the particular Pascal compiler in use arranges the fields within records, thereby losing whatever claim we had on Multilisp's portability. Multilisp solves the problem
by introducing a set of special registers holding the argument values. When CNT_EVLIS has finished evaluation of the arguments all \( N *\text{ARGVAL}'s \) are pulled from the active eval-block and deposited in the registers 1 to N. The remaining registers are set to NIL, thus accounting for missing arguments. The contents of the *FN* blip field are deposited in the special register CURRENT_FN, after which the currently active eval-block is released. The next action depends on the function to be called.

4.4.3 Function Invocation

If the function is a SUBR no further stack manipulation is required. The Functional Object contains a field SUBR_ADDR (see Appendix A) which is an index into a large branch table analogous to the branch table for continuation points. The SUBR is simply called through the branch table. Each SUBR knows that its arguments are located in the global registers. Only if a SUBR needs an eval-block because it in turn is calling EVAL (such as COND, PROG, etc.) it calls SETUP_FRAME which creates the proper frame sequence, using the values in the registers and in CURRENT_FN to assemble the basic frame. It then resets CURRENT_FN to NIL. All other SUBR's simply do their thing and return. If the SUBR returns, a branch is made to the global label mentioned before to pop the Pascal stack. If any SUBR causes an error, the internal error handler checks the contents of CURRENT_FN. If it is not NIL the spaghetti stack is incomplete so the error handler calls SETUP_FRAME to complete it before activating the Interlisp error handler (in this case no
eval-block is created, so that, if control ever returns to this frame it is immediately deleted and control returns to the caller). All SUBR's including EVAL (and therefore all EXPR's as well) return their result in register 1.

If the function is an EXPR then SETUP_FRAME is called immediately to construct the proper frame sequence. In addition an eval-block is created on top of the new frame extension. The *TAIL* blip field of this new block is set to the CDR of the LAMBDA expression of the function (hence, this includes the formal parameter list of the function). The continuation point in the eval-block is set to an index corresponding to the PROGN continuation procedure. Then a branch is made to the main interpreter loop in order to pop the Pascal stack. From there execution continues with a call to the PROGN continuation procedure which repeatedly sets the *TAIL* field of the currently active eval-block to its CDR, and calls EVAL on the CAR of *TAIL* until this is exhausted (Note that the formal parameter list of the LAMBDA expression was skipped the first time the PROGN routine was entered). At this point a call to POP_ACTFRAME releases the currently active frame and a subsequent branch to the main loop continues the interpreter according to the value of the continuation field of the topmost block of the next active frame.

4.4.4 An Example: the Implementation of COND

To illustrate the actions for a blip using function, let us examine the Multilisp implementation of COND:
procedure COND_VM;
(* reg1 = list of clauses *)
begin
    NEW ACTIVATION (nil, CONTINUE_COND, 0);
    (* This creates (a) the basic frame for COND,
        taking the argument (list of clauses) from
        register 1; (b) the frame extension; and (c) a
        block with the specified continuation point. *)
    ACTBLOCK@.BLIP_TAIL := REG(.1.);
    EVAL (CAR (CAR (REG(.1.))));
    (* EVAL will set the BLIP_FORM field of ACTBLOCK
        to its argument. Then returns the value of its
        argument in register 1 if a non-list; otherwise
        the function called leaves its result in
        register 1. *)
end (* cond_vm *);

procedure CNT_COND;
(* This procedure is called only from the main
    interpreter loop. *)
var NEXTFORM : PTR;
begin
    while REG(.l.) = A_NIL do
        (* Result returned from EVAL. *)
        begin
            NEXTFORM := CAR_OF_POP (ACTBLOCK@.BLIP_TAIL);
            (* This sets *TAIL* to CDR (*TAIL*) and
                returns its new CAR if still a list;
                otherwise returns nil. *)
            if NEXTFORM = nil then POP_ACTFRAME;
            (* POP_ACTFRAME deletes the COND frame and
                branches to the main interpreter loop. *)
            EVAL (CAR (NEXTFORM));
        end;
    ACTBLOCK@.BLIP_TAIL := CAR (ACTBLOCK@.BLIP_TAIL);
    (* BLIP_TAIL now points to the clause whose
       condition succeeded. *)
    NEXTFORM := CAR_OF_POP (ACTBLOCK@.BLIP_TAIL);
    (* See if there are any forms in the clause other
       than the condition. *)
    if NEXTFORM = nil then POP_ACTFRAME;
    (* Note that REG(.1.) still contains the value of
       the condition. This will therefore also be the
       result of the COND if this test succeeds. If
       not, proceed with the implicit PROGN. *)
    ACTBLOCK@.CONTINUATION := CONTINUE_PROGN;
    EVAL (NEXTFORM);
end (* cnt_cond *);
procedure CNT NPROG;
(* The continuation point of PROGN *)
var NEXTFORM : PTR;
begin
repeat
  NEXTFORM := CAR OF POP (ACTBLOCK@.BLIP_TAIL);
  if NEXTFORM <> nil then EVAL (NEXTFORM);
  until NEXTFORM = nil;
POP ACTFRAME;
end (* cnt_nprog *);
5. Evaluation

5.1 The Virtual Machine

The construction of the Interlisp Virtual Machine Specification was a truly heroic attempt to capture the essentials of the Interlisp system in a semi-formal way. Many prospective implementors have been misled by the concise function definitions contained in a mere 110-page document. The Multilisp effort has clearly shown, however, that a large body of software is required to realize the VM. And Multilisp, with its 16500 lines of high-level language code, is only a partial implementation (for a list of missing SUBR's, see Appendix A). In spite of its thoroughness, the VM suffers from a number of weaknesses which we shall discuss in this chapter.

The fact that the VM was defined post facto has had a significant influence on its design. It is in general much easier to establish the specification of a new language or software system first, followed by an implementation on the basis of these specifications, after which the two interact in a process of continual refinement, than it is to extract these specifications from an existing system — especially such a large and complex system as Interlisp. In the latter case all too often idiosyncrasies and machine- or system-dependencies are incorporated in the design in order to remain compatible since the investment in human and material resources is too large to allow for major changes in the system itself.
5.1.1 System Dependencies

One area where the VM shows its system dependencies is Input and Output. A very complex I/O mechanism has been specified in the VM which relies heavily on an ASCII environment and a character-oriented file system, particularly that of the TENEX operating system [Bobrow et al., 1972]. Sophisticated user-programmable interrupt facilities are provided which are driven by control characters from an ASCII keyboard. In an IBM environment all one has is a single BREAK or attention-interrupt key which causes an immediate processor interrupt. Moreover, under MTS, terminal input and output is processed in the same manner as file I/O -- both are line-oriented. The user program (e.g., Multilisp) executes a call to the system READ routine after which it is suspended until the READ routine returns a buffer. All terminal editing, such as character- and line-delete, retype or character-insert is handled by the system terminal I/O driver. Attention interrupts do not abort the READ but are stacked until the READ completes. Machine independent Input and Output specifications are probably the greatest challenge to the designer of portable software systems. The Interlisp Virtual Machine needs considerable work in this area before it can be considered portable. Unfortunately, it is generally agreed upon that particularly with respect to I/O increased portability usually means decreased system power. Many systems have powerful interrupt facilities or graphics capabilities which can only be used effectively if system dependencies are allowed.
5.1.2 Complexity

Although some SUBR's, such as IQUOTIENT, can justly be considered minimal instructions of an abstract machine, many functions in the VM are very complex, allowing a great variety of inputs (e.g., stack frame specifications, c.f. section 3.4.3), or performing large amounts of processing (e.g., SETSYNTAX). We believe that the VM can be reduced in size considerably if SUBR's are defined such that they assume at all times the correct number and type of arguments. Many SUBR's check the type of their arguments and provide defaults or perform different actions depending on whether certain arguments are present or not. All these checks and additions can easily be brought to a higher level in the total Interlisp system hierarchy, thus shifting the burden to the compiler.

One particular type check that greatly increases the complexity of the VM is performed on numeric arguments. The VM assumes that no SUBR can be resumed after it has generated an error. The terminology used by the VM Specification is "cause error n with culprit x". The only exceptions are in the definitions of the functions FIX and FLOAT:

```
FIX[n]  If FIXP[n], return n;
          elseif FLOATP[n]:
              Represent and return as an Integer
                  the integer part of n;
          else, FIX[ERRORX[LIST[T0;n]]].
```

The function FLOAT has a similar definition.

ERRORX is not a VM function but is one of the three entries into the Interlisp error package. The VM invokes the error
package through FAULTEVAL if EVAL is called with a bad form, through FAULTAPPLY if a bad Functional Object is applied and through ERRORX in all other cases.

The definition of FIX indicates that it can return an Integer even if an error was caused. If FIX were a SUBR that interacted only with the "outside world" there would be no great difficulty in implementing this (at the cost of some additional programming). The problem lies in the fact that the phrase:

If not FIXP[n], let n be FIX[n].

is one of the most frequently occurring in the VM Specification. Hence, if FIX can return from an error, then every SUBR using FIX must also be able to survive a call to the error processor (and therefore the interpreter). In other words, every SUBR must be in a "clean" state before it can call FIX. This causes a tremendous increase in the complexity of the SUBR encodings. If argument checking were to be lifted to a higher level this problem would simply disappear. For instance, the function IDIFFERENCE is defined in the VM as:

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

This could be replaced with an Interlisp function IDDIFFERENCE

\[
[\text{LAMBDA (I J)} \\
(\%\%IDIFF (\text{FIX} I) (\text{FIX} J)]
\]

using the SUBR definition
%%IDIFF[i;j] Represent and return as an Integer the integer \( i-j \).

There is no reason why the compiler should not be able to produce a good translation for this function assuming that it compiles out bottom level functions such as %%IDIFF. (Using prefix characters such as %% would be quite useful to indicate danger to unskilled programmers -- cf. [Weinreb & Moon, 1979].) Interlisp-10 actually uses this idea in many cases. For instance, FRPLACD is a SUBR in Interlisp-10 which does not check the validity of its arguments; RPLACD is a (compiled) function that checks both arguments and if no error is caused returns FRPLACD on its arguments. Ideally, the VM only contains such "bottom" level functions, making it a much smaller and lower level subset of Interlisp than it currently is. It will be difficult but worth the effort to construct the specifications of such a set of functions in a portable way.

5.1.3 Redundancy

Many SUBR's in the VM are defined strictly in terms of other VM functions. This redundancy causes the implementation of the VM to be substantially larger than it need be unless the implementor executes a SUBR so defined by simply calling the relevant functions (in which case they might as well have been left in LISP and compiled). Some such functions are the mixed mode arithmetic functions but there are many others. For instance, the VM specification of INFILE is

\[
\text{INFILE[file]} \quad \text{Return INPUT[OPENFILE[file;INPUT;OLD]]}.\]
The inefficiency introduced by compiling the corresponding LISP form into a CEXPR is negligible.

5.1.4 Incompleteness

Several VM function specifications are not complete. The most frequent omissions are in the form of omitted "else" clauses such as in CHARACTER (which returns a Character if its argument is a valid character code, but if it is not the action to be taken is not specified).

Another omission is cause for great concern: the lack of a machine-independent assembler. If Interlisp is to be portable then one dire necessity is a portable compiler written in Interlisp which produces intermediate code for an abstract machine, presumably in list format. This machine would be incorporated in the VM (or, probably, a considerably modified version of it). The VM should then provide ASSEMBLE, a SUBR which takes the aforementioned list of abstract machine code as argument, and produces an object of type CEXPR in whatever format the implementor desires or is feasible.

The intermediate code could be assembled "as is" and interpreted by the VM (analogous to the Pascal PCODE interpreter). This would probably be too slow to be economical but still would be considerably faster than interpreted LISP code. It may also be a useful device for machine- and system-independent bootstrapping.

Alternatively, the intermediate code could be translated into directly executable machine code using macros [Griss &
Hearn, 1979], or one could use "threaded coding" [Bell, 1973] where every intermediate code instruction would be translated into a call to a support procedure implementing the actions of the instruction.

If the rest of the Interlisp system makes no assumptions about the specific implementation of either CEXPR objects or the VM, then there is some hope for a completely portable Interlisp system. However, using this approach (as desirable as it may seem at first glance) many questions concerning efficient use of processor, memory and peripheral resources remain to be resolved.

5.2 Multilisp

5.2.1 The Implementation Language

Using Pascal as the implementation language of Multilisp has proved to be quite beneficial. The strict typing rules greatly aided in maintaining consistency of the entire system. The high degree of data abstraction supported by Pascal allowed experimenting with various data and control structures with relative ease. The run-time Debug Package [Jolliffe & Pollack, 1979] available at the University of British Columbia was expected to be of great help in development of the system and analyzing bugs due to addressing interrupts, etc. Actually, little use was made of this facility: once compiled correctly, a module usually ran without further trouble. The ease with which the entire Multilisp system was transported to the VAX 11/780 is
another plus for Pascal as implementation language.

Obviously, Pascal has its drawbacks as well. Particularly, we found the lack of clean pointer arithmetic disturbing. Pascal/UBC has a nice solution for this problem. Generic type transfers are allowed between scalars of any type, hence incrementing a pointer to the next word looks like:

\[ \text{PTRVAL} := \text{ptr} \left( \text{integer (PTRVAL)} + \text{BYTESPERWORD} \right) \]

where "ptr" is the type of PTRVAL. Generic type transfers do not result in extra code — they are no-ops effective at compile-time. Unfortunately, we could not use this feature since we were committed to Standard Pascal. The only type transfer allowed by Standard Pascal is between characters and integers, and the only way to perform pointer arithmetic in Standard Pascal is by record variant overlay, a very unsatisfactory and machine-dependent method. All pointers that never require arithmetic are indeed pointers in Multilisp. All others, primarily those for the Heap and Stack Manager, are declared to be of type ADDRESS, a record with pointer/integer overlay.

A second drawback is that Pascal has no provision for functional variables. Although functional arguments are allowed, and hence formal parameters that "hold" a functional object, this object is not assignable to a declared variable. This would have been useful in that procedures implementing SUBR's as well as continuation procedures themselves could have been deposited in the Multilisp objects of type T_SUBR or
deposited in the continuation fields on the stack, respectively. Instead, we had to use branch tables which causes some slowdown in system performance.

Thirdly, implementing the VM in a high-level language such as Pascal has inevitably led to a significant reduction in speed. The heap and stack support in particular need considerable improvement to become economical. (A small investigation, using a program written in MAYA [Havens, 1978] which makes heavy use of the spaghetti stack facilities, has shown that Multilisp spent roughly 34% of its time in the Stack Manager and another 18% in the Heap Manager and in the functions CAR and CDR.) However, implementing the VM in Assembly Language is prohibitive because of the size of the VM unless large amounts of human and material resources are invested. The resulting implementation would be significantly faster, but would be as un-portable as the current Interlisp-10 system.

5.2.2 Multilisp and the VM

As was stated before, Multilisp is only a partial implementation of the Interlisp Virtual Machine. In particular, many lower level I/O functions have not yet been implemented, and the Terminal Table object is not present for the reasons given in section 5.1.1. Initially, the I/O was redefined to incorporate the line-oriented nature of the MTS operating system and we ended up simulating a line-oriented file system on the VAX-11/780 under VMS which supports character file I/O. The situation is now reversed so that the VAX version is consistent
with the VM Specification. Character file I/O simulation under MTS still remains to be implemented.

Because of the incompatibilities between the VM and the rest of Interlisp stated in section 2.3 we have not been able to load and run significant portions of the Interlisp System. It has therefore not yet been possible to determine the correctness of Multilisp. We can only be (relatively) sure of conformity with the VM Specification. So far as is known to the author, no implementation of the Interlisp System has started with a complete implementation of the Interlisp Virtual Machine and obtained the rest of the system by means of bootstrapping the Interlisp-10 files. So we cannot be sure that it is even possible!

5.3 Directions for Future Research

In order to attain the goal of portability of Interlisp much work remains to be done. First of all, a redesign of the Virtual Machine is desirable in order to remove system dependencies and redundant function definitions. The VM must become a minimal machine containing only functions or instructions which are orthogonal to one another as much as possible.

Secondly, a machine independent interface between a compiler and the VM in the form of intermediate code needs to be designed, so that the large amount of Interlisp software can be run within realistic bounds of time and space.
Thirdly, much of the higher level Interlisp support software needs to be examined and modified in order to remove all the peculiarities due to the Interlisp-10 implementation.

Fourthly, the level on which the Virtual Machine operates is subject to further investigation. The VM is currently defined as a subset of the Interlisp System. Another approach to portability is to define the Virtual Machine in terms of abstract machine instruction codes analogous to the Pascal PCODE machine. Both approaches appear to have merit and a possible solution to this dilemma might lie in a hybrid machine combining the advantages of abstraction at the machine level with the advantages of complete (but minimal) SUBR definitions at the language subset level.

Finally, further investigations are necessary to determine the feasibility and cost-effectiveness of a portable Interlisp system. Both improvements in software design and future advances in hardware capabilities may well offset any initial reductions in speed or increases in space requirements due to portability of the system.
6. Conclusion

In this thesis we have examined the Interlisp Virtual Machine and an implementation thereof in some detail, focusing our attention particularly on portability issues. We have shown in Chapters 3 and 5 that the VM has been a useful device in describing the basics of Interlisp but that it fails in a number of respects. Its system-dependencies, incompleteness of many SUBR definitions, the lack of an intermediate code assembler, and its large size because of much type-checking and default-providing code all conflict with the concept of a portable "virtual" machine.

Chapter 4 described some of the particular difficulties encountered in the implementation of the Virtual Machine and some of the strategies employed in data, control and memory manipulations. Multilisp still needs considerable work to make the implementation of the VM complete and totally compatible with the remainder of the Interlisp System. Chapter 2 has shown in summary that this goal may not even be attainable because of the system dependencies incorporated in the Interlisp bootstrapping files.

We believe that the Abstract Machine approach is a viable way of specifying and implementing a portable Interlisp System. To this end the Virtual Machine needs considerable revision and refinement. It is our hope that the Multilisp implementation may be a guide in isolating those parts of the VM that currently
make it difficult to implement and non-portable. We believe, for reasons set out in Chapter 3, that portability should be of prime concern to all system developers, and particularly to those implementing LISP systems. The development costs may be higher, but the advantages far outweigh the disadvantages.
Bibliography


Appendix A

Functions declared as SUBR's in the VM Specification and as EXPR's in the Interlisp Reference Manual:

ALPHORDER  ANTILOG  ARCCOS
ARCSIN     ARCTAN     ARGLIST
ARRAYSIZE  ARRAYTYP  CALLSSCODE
CLOSEALL   CLOSEF     COPYALL
COPYBYTES  COS        DCHCON
DECLAREDATATYPE  DEFEVAL  DELETECONTROL
DEFILE     DIFFERENCE  DISMISS
DISPLAYTERMP  DOBE      DRIBBLE
DUNPACK     ECHOCONTROL EXPT
FDIFFERENCE  FETCHFIELD FILEPOS
FIX         FIXP       FLESSP
FLOAT       FMINUS     FNTYP
FULLNAME    FUNCTION   GCD
GCCGAG      GETDESCRIPTIONS GETFIELDSPECS
GETFILEINFO  GETPROPLIST GETSYNTAX
HARRAYP     HARRAYSIZE  IDATE
IDIFFERENCE  ILESPP     IMINUS
LESSP       LOG        LRSH
MAKEBITTABLE  MAPHASH   NARGS
NCREATE     OPENFILE   RAND
RANDACCESSSP  RANDESET  RECLAIM
RELSTKP     RENAMEFILE REPLACEFIELD
RPLACA      RPLACD     RSH
SETA        SETFILEINFO SETPROPLIST
SETSYNTAX   SIN        SKREAD
SMALLP      SQRRT      STORAGE
STRPOS      STRPOSL   TAN
USERDATATYPES  USERNAME

Functions declared as SUBR's in the Virtual Machine Specification which do not appear in the Interlisp Reference Manual:

BUFP  CHANGECCODE  GETINTERRUPT
MKFRAME  OVERFLOW  SETALINK
SETCLINK  SETINTERRUPT
Functions declared as SUBR's in the Interlisp Reference Manual which do not appear in the VM Specification:

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAYBEG</td>
<td>BLKAPPLY</td>
<td>BLKAPPLY*</td>
</tr>
<tr>
<td>CLOSER</td>
<td>DDT</td>
<td>ELTD</td>
</tr>
<tr>
<td>ERRORN</td>
<td>ERRORSET</td>
<td>ERRORSTRING</td>
</tr>
<tr>
<td>ERROR!</td>
<td>ESCAPE</td>
<td>FCHARACTER</td>
</tr>
<tr>
<td>FRPLACA</td>
<td>FRPLACD</td>
<td>GCMESS</td>
</tr>
<tr>
<td>GETATOMVAL</td>
<td>GETBLK</td>
<td>HERALD</td>
</tr>
<tr>
<td>INTERRUPTABLEP</td>
<td>JSYS</td>
<td>LOC</td>
</tr>
<tr>
<td>MAKESYS</td>
<td>NCONC</td>
<td>NTYP</td>
</tr>
<tr>
<td>OPENF</td>
<td>OPENR</td>
<td>OPNJFN</td>
</tr>
<tr>
<td>PACK*</td>
<td>RELBLK</td>
<td>RESET</td>
</tr>
<tr>
<td>RESETVARS</td>
<td>RESUME</td>
<td>SCODEP</td>
</tr>
<tr>
<td>SETATOMVAL</td>
<td>SETERRORN</td>
<td>SETSBSIZE</td>
</tr>
<tr>
<td>SETSTKNAME</td>
<td>SWPARRAY</td>
<td>SWPARRAYP</td>
</tr>
<tr>
<td>SYSTEMTYPE</td>
<td>VAG</td>
<td></td>
</tr>
</tbody>
</table>

Interlisp Virtual Machine SUBR's not yet implemented in Multilisp:

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>BACKTRACE</td>
<td>BKLINBUF</td>
<td>BKSYSBUF</td>
</tr>
<tr>
<td>BUFP</td>
<td>CALLSCCODE</td>
<td>CHANGECCODE</td>
</tr>
<tr>
<td>CLEARBUF</td>
<td>CONTROL</td>
<td>COPYBYTES</td>
</tr>
<tr>
<td>COPYTERMTABLE</td>
<td>DELETECONTROL</td>
<td>DISPLAYTERMP</td>
</tr>
<tr>
<td>DOBE</td>
<td>DRIBBLE</td>
<td>DRIBBLEFILE</td>
</tr>
<tr>
<td>ECHOCONTROL</td>
<td>ECHOMODE</td>
<td>FILEPOS</td>
</tr>
<tr>
<td>GCD</td>
<td>GCTRP</td>
<td>GETFILEINFO</td>
</tr>
<tr>
<td>GETINTERRUPT</td>
<td>GETTERMTABLE</td>
<td>IDATE</td>
</tr>
<tr>
<td>INTERRUPTABLE</td>
<td>LINBUF</td>
<td>PRIN3</td>
</tr>
<tr>
<td>PRIN4</td>
<td>RAISE</td>
<td>RANDACCESSP</td>
</tr>
<tr>
<td>RATEST</td>
<td>RATOM</td>
<td>READP</td>
</tr>
<tr>
<td>RESETTERMTABLE</td>
<td>RSTRING</td>
<td>SETFILEINFO</td>
</tr>
<tr>
<td>SETINTERRUPT</td>
<td>SETTERMTABLE</td>
<td>SKREAD</td>
</tr>
<tr>
<td>SYSBUF</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix B

Multilisp Object Representation:

PTR = @ OBJECT;

OBJECT = packed record
    case OBJTYPE : OBJECT_TYPE of
      T_INDIRECT : (INDPTR : PTR);
      T_CMPCT_NIL,  
      T_CMPCT_NEXT : (CARPTR : PTR);
      T_LISTP : (CARP, CDRP : PTR);
      T_FORWARD : (FWDPTR : PTR);
      T_LITATOM : (TOPVAL,
                    PLIST,
                    FUNDEF,
                    HASH_LINK : PTR;
                    PNAME : STRING);
      T_FIXP : (IVAL : INTEGER);
      T_FLOATP : (RVAL : REAL);
      T_IARRAYP : (IR_SIZE : SHRTINT;
                    IR_VECTOR : INTVECTOR);
      T_PARRAYP : (PR_SIZE : SHRTINT;
                    PR_VECTOR : PTRVECTOR);
      T_HARRAYP : (HR_SIZE : SHRTINT;
                    HR_VECTOR : BINDVECTOR);
      T_READTABLEP : (MACROS_ENABLED : BOOLEAN;
                       READ_TABLE : RDTBLVECTOR);
      T_STACKP : (FRAMEP : STACKPTR;
                   STP_LINK : PTR);
      T_STRINGP : (FIRST_CHAR,
                    CHAR_COUNT : SHRTINT;
                    SOURCE : PTR);
      T_STRINGPNAME : (STR : STRING);
Appendix B

```
T_BITTABLEP : (BIT_TABLE : BITTBLVECTOR);

T_USER_TYPE : (TYPESIZE : SHRTINT;
               TYPENAME,
               LAST_DESCR,
               NEXT_TYPE : PTR);

T_USER_OBJECT : (USERTYPE : PTR;
                 BYTES : LINE);

T_DESCR_PTR,
T_DESCR_BIT,
T_DESCR_FIXP,
T_DESCR_FLOATP : (OFFSET : SHRTINT;
                   PREV_DESCR : PTR);

T_CEXPR,
T_SUBR : (NPARAMS : COUNTER;
           EVALFLAG,
           SPREADFLAG : BOOLEAN;
           case OBJECT_TYPE of
               T_CEXPR : (); (* no such thing yet *)
               T_SUBR : (SUBR_ADDR : SUBR_INDICES));

end (* object *);
```

Memory requirements in bytes

<table>
<thead>
<tr>
<th></th>
<th>Pascal/UBC (Amdahl 470)</th>
<th>VMS Pascal (VAX 11/780)</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_ARRAYP</td>
<td>4;</td>
<td>3;</td>
</tr>
<tr>
<td>M_BITTBLP</td>
<td>516;</td>
<td>33;</td>
</tr>
<tr>
<td>M_CEXPR</td>
<td>8;</td>
<td>3;</td>
</tr>
<tr>
<td>M_COMPACT</td>
<td>8;</td>
<td>5;</td>
</tr>
<tr>
<td>M_DESCR</td>
<td>8;</td>
<td>7;</td>
</tr>
<tr>
<td>M_FIXP</td>
<td>8;</td>
<td>5;</td>
</tr>
<tr>
<td>M_FLOATP</td>
<td>16;</td>
<td>5;</td>
</tr>
<tr>
<td>M_FORWARD</td>
<td>8;</td>
<td>5;</td>
</tr>
<tr>
<td>M_LISTP</td>
<td>12;</td>
<td>9;</td>
</tr>
<tr>
<td>M_LITATOM</td>
<td>22;</td>
<td>19;</td>
</tr>
<tr>
<td>M_RDTBLP</td>
<td>1028;</td>
<td>1026;</td>
</tr>
<tr>
<td>M_SMALLP</td>
<td>2;</td>
<td>1;</td>
</tr>
<tr>
<td>M_STACKP</td>
<td>12;</td>
<td>9;</td>
</tr>
<tr>
<td>M_STRINGP</td>
<td>12;</td>
<td>9;</td>
</tr>
<tr>
<td>M_STRPNAME</td>
<td>4;</td>
<td>3;</td>
</tr>
<tr>
<td>M_SUBR</td>
<td>12;</td>
<td>5;</td>
</tr>
<tr>
<td>M_TERMTBLP</td>
<td>8;</td>
<td>5;</td>
</tr>
<tr>
<td>M_USERTYPE</td>
<td>16;</td>
<td>15;</td>
</tr>
<tr>
<td>M_USEROBJ</td>
<td>8;</td>
<td>5;</td>
</tr>
</tbody>
</table>

Appendix C

The algorithm for non-recursive list compaction used by the Multilisp Garbage Collector:

procedure MOVE_LIST (var OLDLIST : PTR);

(* Iterative list move and compaction procedure. *)
(* Will also work on circular lists. *)
(* On exit OLDLIST is reset to address of copied list *)

label 1;

(* for fast CDR'ing down the list *)

var CARVAL, LIST, NEXT : PTR;

begin
LIST := OLDLIST;  (* local to CDR down OLDLIST *)
OLDLIST := HEAP_BOTTOM.P;  (* and set var parameter *)

L: if HEAP_BOTTOM.I >= HEAP_TOP.I
then ERROR (1, nil);
CARVAL := CAR (LIST);  (* NOTE: Not copied yet! *)
NEXT := CDR (LIST);  (* Jumps thru indirect cells *)
LIST@.OBJTYPE := T_FORWARD;  (* leave forwarding *)
LIST@.FWDPTR := HEAP_BOTTOM.P;  (* address behind *)

with HEAP_BOTTOM.P@ do
   if NEXT@.OBJTYPE <= T_LISTP then
      begin
         OBJTYPE := T_CMPCT_NEXT;
         CARPTR := CARVAL;
         HEAP_BOTTOM.I := HEAP_BOTTOM.I + M_COMPACT;
         LIST := NEXT;
         goto 1;  (* and CDR down the old list *)
      end
   else if NEXT = A_NIL then
      begin
         OBJTYPE := T_CMPCT_NIL;
         CARPTR := CARVAL;
         HEAP_BOTTOM.I := HEAP_BOTTOM.I + M_COMPACT;
      end
   else (* NEXT points to an already copied list or *)
      begin (* to some non-list, so make a true listp *)
         OBJTYPE := T_LISTP;
         CAR := CARVAL;
         CDRP := NEXT;
         HEAP_BOTTOM.I := HEAP_BOTTOM.I + M_LISTP;
      end
   end (* move_list *);
Appendix D

Stack frame declarations:

STACKPTR = @ STACKFRAME;
BLOCKPTR = @ STACKBLOCK;

STACKFRAME = packed record
  LAST,
  NEXT,
  TOP : ADDRESS;

    case BOOLEAN of
      TRUE : (NAME : PTR;
               REFCNT,
               NARGS : COUNTER;
               ARGP : BINDVECTOR);

      FALSE : (USECNT : COUNTER;
                PROGFLAG,
                TRACEMARK : BOOLEAN;
                MACROSTATUS : MACROSTATE;
                ALINK,
                BLINK,
                CLINK : STACKPTR;
                TOP_BLOCK : BLOCKPTR);
    end (* stackframe *);

STACKBLOCK = packed record
  LAST_BLOCK : BLOCKPTR;
  BLIP_FORM,
  BLIP_TAIL : PTR;
  BLOCKSIZE : COUNTER;
  CONTINUATION : CNT_POINTS;

    case BLIP_ARGS : BOOLEAN of
      TRUE : (NARGVALS : COUNTER;
               BLIP_FN : PTR;
               ARGVÄL : PTRVECTOR);

      FALSE : (DOTFLAG,
                BRACKETFLAG : BOOLEAN;
                READFILE : FILEINDEX;
                NTEMPS : COUNTER;
                TEMPINT : SHRTINT;
                TEMP : PTRVECTOR);
    end (* stackblock *);
Appendix D

Graphic layout for stack frames under VMS Pascal:

--- Basic frame layout: (size: 18 + 8 * NARGS bytes)

| LAST |
| NEXT |
| TOP |
| NAME |
| REFCNT | NARGS |

--- Frame extension layout: (size: 30 bytes)

| LAST |
| NEXT |
| TOP |
| USECNT | P | T | M |
| ALINK |
| BLINK |
| CLINK |
| TOP BLOCK |

where P: PROGFLAG, T: TRACEMARK, M: MACROSTATUS, ---: unused
Appendix D

=== Blip using block layout:  (size : 19 + 4 * NARGVALS bytes)

*+---------------------------------+*+---------------------------------+*
| LAST_BLOCK                        |
*+---------------------------------+*+---------------------------------+*
| BLIP FORM                         |
*+---------------------------------+*+---------------------------------+*
| BLIP TAIL                         |
*+---------------------------------+*+---------------------------------+*
| BLOCKSIZE | CONT'N  B|==| NARGVALS   |
*+---------------------------------+*+---------------------------------+*
| BLIP_FN   |
*+---------------------------------+*+---------------------------------+*

where CONT'N : CONTINUATION,  B : BLIP_ARGS

then for each *ARGVAL* I :

*+---------------------------------+*+---------------------------------+*
| ARGVAL(.I.)                      |
*+---------------------------------+*+---------------------------------+*

=== Temporaries block layout:  (size : 18 + 4 * NTEMPS bytes)

*+---------------------------------+*+---------------------------------+*
| LAST_BLOCK                        |
*+---------------------------------+*+---------------------------------+*
| BLIP FORM                         |
*+---------------------------------+*+---------------------------------+*
| BLIP TAIL                         |
*+---------------------------------+*+---------------------------------+*
| BLOCKSIZE | CONT'N  B|D|F| READFILE   | NTEMPS   |
*+---------------------------------+*+---------------------------------+*
| TEMPINT   |
*+---------------------------------+*+---------------------------------+*

where B : BLIP_ARGS,  D : DOTFLAG,  F : BRACKETFLAG

then for each temporary:

*+---------------------------------+*+---------------------------------+*
| TEMP(.I.)                         |
*+---------------------------------+*+---------------------------------+*
Appendix E

The infinite loop of the interpreter:

begin (* Multilisp Main Program *)

(* Initialize the system *)

1111 : (* Label for re-entering the loop from ERROR, etc. *)

if ATTN_INTERRUPT then ERROR (18, A NIL);
if ACTBLOCK = nil then POP_ACTFRAME;
if ACTFRAME@.USECNT > 1 then
  if ACTFRAME <> TOPFRAME then SPLIT_ACTFRAME;
case ACTBLOCK@.CONTINUATION of
  DONT CONTINUE : REL_ACTBLOCK;
  CONTINUE TOPLEVEL : CNT_TOPLEVEL;
  CONTINUE_AND : CNT_AND;
  CONTINUE_ARG : CNT_ARG;
  CONTINUE_COND : CNT_COND;
  CONTINUE_1_EVALQT : CNT_1_EVALQT;
  CONTINUE_2_EVALQT : CNT_2_EVALQT;
  CONTINUE_EVLIS : CNT_EVLIS;
  CONTINUE_1_PROG : CNT_1_PROG;
  CONTINUE_2_PROG : CNT_2_PROG;
  CONTINUE_MAPATOMS : CNT_MAPATOMS;
  CONTINUE_MAPHASH : CNT_MAPHASH;
  CONTINUE_OR : CNT_OR;
  CONTINUE_PROG1 : CNT_PROG1;
  CONTINUE_PROGN : CNT_PROGN;
  CONTINUE_1_READ : CNT_1_READ;
  CONTINUE_2_READ : CNT_2_READ;
  CONTINUE_3_READ : CNT_3_READ;
  CONTINUE_4_READ : CNT_4_READ;
  CONTINUE_5_READ : CNT_5_READ;
  CONTINUE_6_READ : CNT_6_READ;
  CONTINUE_REHASH : CNT_REHASH;
  CONTINUE_SETARG : CNT_SETARG;
  CONTINUE_SETN : CNT_SETN;
  CONTINUE_SETQ : CNT_SETQ;
end;

goto 1111;

9999 : (* Exit: got here through LOGOUT *)

end (* Multilisp Main Program *).
# $run intr:vm
> Welcome to Multilisp - version 2.0 - 01:35 / 05-07-80
>
> ... ; quote (') setup, CLEARSTKLST <- NIL
>
> (PUTDQ FOO
>  (NLAMBDA (STP)
>   (PRINT 'Hi)
>   ((LAMBDA (FRAME)
>     (COND ((STACKP FRAME) (SET STP FRAME))
>     (T (PRINT FRAME)))
>     (STKPOS 'FOO))
>   (PRINT 'there)
>   'FOO-exit))
>
> FOO
>  (DEBUG STACK)
>  STACK
>
> + <<<<<< Basic created at #0065505C EVALQT
> + <<<<<< Frame created at #00655078
> + <<<<<< Block created.
>   ; call EVALQT
> + <<<<<< Block created.
>   ; for running EVALQT
> + <<<<<< Block created.
> + <<<<<< Block created.
> + <<<<<< Releasing block.
> + <<<<<< Basic created at #006550B4 FOO
> + <<<<<< Frame created at #006550D0
> + <<<<<< Block created.
> + <<<<<< Block created.
> + <<<<<< Releasing block.
> + <<<<<< Block created.
> + <<<<<< Block created.
> + <<<<<< Block created.
> + <<<<<< Releasing block.
> + <<<<<< Basic created at #00655124 STKPOS
> + <<<<<< Frame created at #00655158
> + <<<<<< Releasing frame at #00655158
> + <<<<<< Releasing basic at #00655124 STKPOS

Appendix F

Annotated trace of stack example FOO
Appendix F

+ <<<<<< Splitting frame to #00655124; reactivate FO0
+ <<<<<< Releasing block.; LAMBDA arg Eval'ed
+ <<<<<< Basic created at #00655160 (LAMBDA (FRAME) --)
+ <<<<<< Frame created at #0065517C
+ <<<<<< Block created.; for running LAMBDA
+ <<<<<< Basic created at #006551B8; call COND
+ <<<<<< Block created.; Eval lst clause
+ <<<<<< Block created.; Eval STACKP arg
+ <<<<<< Releasing block.; STACKP arg Eval'ed
+ <<<<<< Releasing block.; call STACKP
+ <<<<<< Releasing block.; exit STACKP
+ <<<<<< Block created.; Eval SET args
+ <<<<<< Releasing block.; SET args Eval'ed
+ <<<<<< Releasing block.; call SET
+ <<<<<< Releasing block.; exit SET
+ <<<<<< Releasing frame at #006551D4; exit COND
+ <<<<<< Releasing basic at #006551B8; exit LAMBDA
+ <<<<<< Releasing frame at #0065517C; exit STACKP
+ <<<<<< Block created.; Eval PRINT arg
+ <<<<<< Releasing block.; PRINT arg Eval'ed
+ <<<<<< Releasing block.; call PRINT

> there
+ <<<<<< Releasing frame at #00655124; exit PRINT
+ <<<<<< Splitting frame to #00655124; reactivate EVALQT
+ <<<<<< Releasing frame at #00655124; exit FO0
+ <<<<<< Basic created at #006551B8; reactivaTE EVALQT
+ <<<<<< Frame created at #0065517C; for running EVALQT
+ <<<<<< Block created.; call PRINT
+ <<<<<< Block created.; for running EVALQT
+ <<<<<< Block created.; read a list
+ <<<<<< Block created.; read a list
+ <<<<<< Releasing block.; list done
+ <<<<<< Releasing block.; list done
+ <<<<<< Releasing block.; exit READ
+ <<<<<< Block created.; Eval RETTO args
+ <<<<<< Releasing block.; RETTO args Eval'ed
+ <<<<<< Releasing block.; call RETTO
+ <<<<<< Basic created at #0065517C RETTO
+ <<<<<< Frame created at #006551A8
+ <<<<<< Releasing frame at #006551A8; release active chain
+ <<<<<< Releasing basic at #0065517C RETTO
Appendix F

+ <<<<> Releasing frame at #00655140 ; release active chain
  <<<<> Releasing basic at #00655124 EVALQT
  <<<<>evaluating frame at basic
  <<<<> Releasing block.
+ <<<<> Splitting frame to #00655124 FOO
  <<<<> Releasing frame.
  <<<<> Releasing basic at #00655124 LAMBDA arg Eval'ed
  <<<<> Block created.
  <<<<> Frame created at #0065517C for running LAMBDA
  <<<<> Block created.
+ <<<<> Basic created at #006551B8 COND
  <<<<> Block created.
  <<<<> Frame created at #006551D4
+ <<<<> Block created.
  <<<<> Block created.
  <<<<> Releasing block.
+ <<<<> Block created.
+ <<<<> Releasing block.

> Hello

+ <<<<> Releasing frame at #006551D4 ; exit PRINT
  <<<<> Releasing basic at #006551B8 COND
  <<<<> Releasing frame at #0065517C ; exit LAMBDA
  <<<<> Block created.
  <<<<> Releasing block.
+ <<<<> Block created.
+ <<<<> Releasing block.

> there

+ <<<<> Releasing frame at #00655124 ; exit FOO
  <<<<> Splitting frame to #00655124 EVALQT
  <<<<> Releasing frame at EVALQT
  <<<<> Basic created at #00655124 (LAMBDA (FRAME) --)
  <<<<> Frame created at #00655140 ; for running EVALQT
  <<<<> Block created.
+ <<<<> Block created.
+ <<<<> Releasing block.

* (LOGOUT)

+ <<<<> Block created.
+ <<<<> Releasing block.

> Bye! (34 fun calls, 14 bframes, 18 frame exts, 60 blocks)