Implementation of Interlisp on the VAX[†]

Raymond L. Bates, David Dyer and Johannes A. G. M. Koomen[††]

University of Southern California
Information Sciences Institute
4676 Admiralty Way
Marina del Rey, California 90291

1. Introduction

This paper presents some of the issues involved in implementing Interlisp [19] on a VAX computer [24] with the goal of producing a version that runs under UNIX [17], specifically Berkeley VM/UNIX. This implementation has the following goals:

- To be compatible with and functionally equivalent to Interlisp-10.
- To serve as a basis for future Interlisp implementations on other mainframe computers. This goal requires that the implementation to be portable.
- To support a large virtual address space.
- To achieve a reasonable speed.

The implemention draws directly from three sources, Interlisp-10 [19], Interlisp-D [5], and Multilisp [12]. Interlisp-10, the progenitor of all Interlisps, runs on the PDP-10 under the TENEX [2] and TOPS-20 operating systems. Interlisp-D, developed at Xerox Palo Alto Research Center, runs on personal computers also developed at PARC. Multilisp, developed at the University of British Columbia, is a portable interpreter containing a kernel of Interlisp, written in Pascal [9] and running on the IBM Series/370 and the VAX. The Interlisp-VAX implementation relies heavily on these implementations. In turn, Interlisp-D and Multilisp were developed from *The Interlisp Virtual Machine Specification* [15] by J Moore (subsequently referred to as the VM specification), which discusses what is needed to implement an Interlisp by describing an Interlisp Virtual Machine from the implementors' point of view. Approximately six man-years of effort have been spent exclusively in developing Interlisp-VAX, plus the benefit of many years of development for the previous Interlisp implementations.

2. History of the Project

A few years ago the research community ceased to consider Interlisp-10 a useful research vehicle because of its limited address space. A search began to provide a new LISP environment powerful enough to support current and future research. There was considerable discussion of abandoning the Interlisp dialect entirely in favor of Maclisp [14], LISP Machine LISP [25], NIL [26], or Common LISP. The choice of LISP dialect would to some extent dictate the choice of hardware. Potentially attractive hardware were the CADR [11] (MIT LISP Machines) and Xerox 1100 Scientific Information Processors (Interlisp-D machines, also known as Dolphins or D0's). Both are *personal* LISP machines. Also considered were machines not specifically oriented toward LISP. They included the PERQ and the PRIME (both personal machines), as well as the M68000-based personal machines, which were promised to be available "soon." The high cost and unpredictable future of each of these personal machines were strong influences against their selection. The new feature of extended addressing on TOPS-20 was also considered and rejected as the basis for a new LISP implementation on the PDP-10.

The DEC VAX computer was selected as the machine to host the new Interlisp for several reasons. It has become an extremely popular machine, especially for universities and research facilities.

[†] The development of Interlisp-VAX was sponsored by the Defense Advanced Research Projects Agency under contract number MDA 903-81-C-0335. Views and conclusions in this report are the authors' and should not be interpreted as representing the official opinion or policy of the Defense Advanced Research Projects Agency, the U.S. Government, or any other person or agency connected with them.

[††] Current affiliation: Department of Computer Science, University of Rochester, Rochester, NY 14627

Although each of the alternative hardwares has acquired a user community, none approaches the popularity of the VAX. The VAX family of computers promises to have a long life, to be widely available, to be extensively supported, and to have a wide variety of price and performance ranges. It is anticipated that the family will be extended both up in performance and down in price. All of these characteristics enhance the usefulness and longevity of Interlisp-VAX compared to the alternatives.

In June 1980 serious work began on the development and implementation of an Interlisp compatible with the VAX series of computers. Initially, most of the effort was directed at the planning and detailed design of the implementation of various critical parts. By the end of the year, the writing of code specific to Interlisp-VAX was begun. Using the Multilisp system as a template, a new Interlisp kernel was developed in the language C [10]. In parallel, the existing Interlisp compiler was modified to produce VAX code. Both of these tasks were essentially completed by August 1981. Since the beginning of 1981, various parts of the existing Interlisp code have been adapted or rewritten to fit the VAX-UNIX mold. Currently the project is substantially completed. The first release of the Interlisp-VAX system was made publicly available in March 1982.

3. Basic Design Decisions

After the initial choices of a target machine and the dialect of LISP, a multitude of choices remained. Foremost were the overall implementation strategy and the implementation language for that part of the system (if any) that was not to be written in Interlisp. Although most of Interlisp is written in itself, another language is traditionally used for a small kernel of code to implement the primitives that are difficult or impossible to implement in LISP.

The resulting implementation follows traditional LISP implementation techniques more closely than do some other new LISP implementations. For example, it does not use CDR coding, or any custom microcode; nor does it require any other special hardware. To some extent, this reflects our design goal that our implementation be portable to other hardware and that it run on VAXes not dedicated to or specially modified for Interlisp. It was also important to minimize the uncertainty that the resulting system would run and be usable. Unproven or experimental techniques were never seriously considered.

3.1 Implementation Language

A LISP implementation written entirely in itself has proven to be viable and was considered for Interlisp-VAX. Ultimately this approach was rejected because of the anticipated difficulty of bootstrapping, and the uncertainty that implies, and because the lowest level LISP code in such a system would likely be very machine dependent, and so not an advantage over a conventional implementation language.

The implementation languages considered were C, Pascal, BLISS [1], and assembly language. The importance of the choice of implementation language varies inversely with the amount of code to be written in LISP. The three primary considerations in the choice were cost in programming time, efficiency of resulting object code, and portability to other machines.

An implementation written entirely in assembly language, such as Interlisp-10, was eliminated because of its lack of portability and its prohibitive cost in programming time. It was decided that writing a small amount of the most critical code in machine language would gain most of the efficiency advantages of an implementation written entirely in machine language at a fraction of its cost. This has proven to be the case. Measurements show that the portion of time spent in code written in C is small.

BLISS would have been a good choice for the VAX, but it is not available on non-DEC machines and is not yet available on UNIX. It did not appear likely that BLISS would be available on non-DEC machines in the future.

Pascal and C are both suitable languages from the viewpoints of portability and availability. Since UNIX was chosen as the host operating system for our VAX, C was the clear favorite. But Pascal also had its advantages, the primary one being the availability of the Multilisp program, which met major portions of the Interlisp VM specification. Considering that the language would be used as an implementation language, Pascal was clearly inferior for the project's purposes. Pascal was not originally intended as an implementation language. It does not easily allow the kinds of data manipulations necessary for efficiency and ease of expression. C ultimately was chosen over Pascal because of its position as the universal implementation language for UNIX and the growing popularity of UNIX as an operating system.

3.2 Division of Labor Among Languages

The amount of code to be written in Interlisp versus that to be written in C or assembly language was another factor. As much code as practical was written in Interlisp without unduly complicating the process of bootstrapping and debugging, or making the LISP code unduly complex or machine dependent. The ratio between Interlisp and C code for Interlisp-VAX is similar to the ratio between Interlisp and machine language in Interlisp-10, except that the services provided by TENEX or TOPS-20 in Interlisp-10 are mostly written in LISP. Interlisp-VAX relies greatly on Interlisp-D, because Interlisp-D implements Interlisp almost completely in

Interlisp (with a minute kernel written in BCPL [16] and microcode). The availability of the actual Interlisp code to implement TENEX or TOPS-20 compatibility was important in achieving a successful implementation.

The Interlisp-VAX interface to the UNIX operating system is simple. Little is required beyond the basic attributes of reading and writing data, delivering interrupts from terminal input, and providing address space. Except for file names and "operating system compatibility," problems should be minor for future implementations based on Interlisp-VAX. The Interlisp model of a file name coincides with a TENEX or TOPS-20 file name, thus it is not compatible with most other operating systems.

Much low-level code is required to map TENEX's or TOPS-20's complex implementation of version numbers and long file names, which Interlisp implicitly depends upon, into UNIX's short, unadorned file names. Our original implementation has since been reworked to encompass VMS's somewhat different file names, and will have to be modified again later if UNIX file names are changed. File names' dependency has also proven to be one of the more annoying glitches to users, who frequently have "canned" directory names from TENEX or TOPS-20 that cannot be mapped directly into UNIX directory structures.

In addition to the low-level operating system interface, the Interlisp-VAX kernel contains basic memory management, spaghetti stack primitives [3], a garbage collector, and the interpreter. LISP code was written or acquired to provide arrays and hash arrays, datatypes, all terminal support above the level of the raw get character and the raw put character, file I/O above the level of the read block and the write block, file name recognition, and all of LISP's READ and PRINT operations.

For successful division of labor between LISP and C, two slightly unusual features were essential. First, the compiler for the VAX was developed and debugged using Interlisp-10. Second, the C kernel contains a throwaway *simple* version of LISP's READ and PRINT, which simplified the debugging and bootstrapping processes.

Almost all of the Interlisp system is required for the compiler to work. Although the compiler was one of the last components of Interlisp-VAX to be brought up on the VAX, compiled LISP code was used almost from the beginning. Since the Interlisp system uses advanced language features of Interlisp (CLISP, Records, etc.), it cannot be interpreted or compiled in anything less than a complete Interlisp system. Therefore, a cross compiler for a new machine is necessary. Interlisp-D and Interlisp-VAX were implemented using cross compilers. Multilisp developed from the VM specification but, without the support of a cross compiler, has not bridged the functionality gap necessary to load the Interlisp environment.

The process of bootstrapping was accelerated by having a primitive READ, EVAL, and PRINT loop built into the kernel. Initially the usual C debuggers were used to start the READ, EVAL, and PRINT working. Later the VM was used to debug itself.

The current Interlisp-VAX kernel contains approximately 1000 lines of assembly language devoted to function linkage, free variable lookup, and low level interrupt handling, as well as a little over 11,000 lines of C code providing basic memory management, spaghetti stack primitives, a garbage collector, the interpreter and a low-level operating system interface. Of the approximately 83000 lines of Interlisp code in this implementation, about 16000 were written explicitly for this implementation. About 9500 are shared with Interlisp-D, and about 57500 are shared with the other Interlisp implementations.

4. Overall Memory Management

4.1 Memory Management Decisions

Interlisp-VAX uses a *BBOP* (big bag of pages) memory management scheme, where the "page" is 64K bytes. This scheme was selected over a *tagged object* architecture or a *tagged pointer* architecture, because the VAX is not a tagged architecture machine. It would not use tagged objects or pointers efficiently. The greatest advantages of a *BBOP* memory management schemes are its simplicity, efficient use of space and efficiency of data management. The architecture of all existing existing machines allows blocks of storage to be allocated efficiently within the block of space. No special hardware is required.

Similar arguments influenced choosing to segment addresses by pages rather than to partition the address space as a whole. Chunks of address space are easily found, whereas the overall shape and texture of the address space varies widely among machines. Likewise, a segmentation appropriate for a 4-megabyte system would not be appropriate for a 40-megabyte system, and we wanted Interlisp-VAX to scale up smoothly. For the VAX, with a 32-bit address space and a 32-bit pointer, 64K bytes was the logical size, since 64K bytes is the square root of the address space. Two of these "sectors" are devoted to a table of data type numbers that serve as indices to a third sector containing multiword descriptors of each data type. These three sectors are the only fixed-memory allocations in the system. All other storage for user-defined and predefined data types is allocated from the operating system pool.

All objects in a 64K block of data are of a single type. For areas containing variable length objects, either those objects will contain no pointers themselves (e.g., PNAMES) or the pointer is to a *sequence descriptor* that describes the object. The basic datatypes definition mechanisms allow for all combinations of datatypes -- fixed length, variable length, those containing pointers, and those not containing pointers.

4.2 Garbage Collection

The Interlisp-VAX kernel contains a nonrecursive, copying garbage collector based on Cheney's Algorithm [6]. The practicality of this scheme for very large address space remains to be proven. Garbage collection, although infrequent, is expensive. For Interlisp-VAX a compacting collector achieves locality of reference in the expected large virtual address space. Although copying is the simplest and most efficient method of compacting, its main disadvantage is the requirement that the operating system, during collection, provide twice as much working storage as is being used. Another disadvantage of the scheme is that it is necessary to be certain of what is and what is not a pointer. In a traditional mark and sweep collection, any questionable objects can be treated as pointers, and at worst, some space will not be collected. In a copying collector, all pointers and only pointers must be moved. Because of this hazard Interlisp-VAX does not contain a "VAG" operator, which converts a random integer into a pointer.

Interlisp-VAX uses approximately 4-megabytes of virtual address space at startup, which compares to UNIX's default restriction of 6-megabytes per process. The 6-megabyte limit is rarely reached during a normal day's session. Only when a massive computation is in progress is collection necessary; and then it is extremely slow, taking several minutes elapsed time (though only a few seconds cpu time!). If the system limit is increased to 11-megabytes, those same compilations run to completion without garbage collection. We believe that the lack of garbage collection does not affect the paging rate significantly.

Our experience has shown that the garbage collector is not an important part of the overall efficiency of the implementation, provided one follows Jon L White's dictum: "Don't do it" [27]. The best overall performance tradeoff is to increase the amount of virtual address space in use. Garbage collect only as a result of limitations in the operating system or hardware architecture.

5. Representation Decisions

5.1 Stack Representation

The choice of representation for LISP data structures is crucial to the ultimate efficiency of a LISP implementation. The greatest compromises in this area were made in favor of efficiency at the expense of portability. As few different structures as practical were used in order to keep the number of different access methods to a minimum.

The stack representation is the most complex, the most tuned to the VAX, and the most vital to efficient operation. The stack representation of Interlisp-VAX uses the VAX's native instructions *CALLS* to construct stack frames and *RET* to destroy them. The auxiliary mechanisms to map the VAX's stack into the more complex spaghetti stack expected by Interlisp are the most complex code in the kernel. "Only" 10 to 15 percent of the time is spent in function application and stack management code. The details of the stack management are peculiar to the VAX and will need to be substantially redone to achieve efficiency in another implementation. The implementors believe the choice of efficiency above all else for stack representation is justified.

5.2 Binding Scheme

Interlisp-10 has employed two different binding schemes for variables. Currently it uses a *shallow*-binding scheme. Prior to April 1976 it used a *deep*-binding scheme. In both shallow binding and deep bindings, associated with each variable there is a special cell (the *value cell*), which normally contains the the top level value of the variable. Using deep binding, when a variable is rebound, a *name/new-value* pair is stored on the stack. To obtain or modify the current value of a variable, the stack must be searched to determine whether or not the variable was rebound. This is potentially time consuming especially if the stack is large. At the time of unbinding the variable or spaghetti context switch, no special actions are required. Under a shallow-bound system, the current value of a variable is always stored in the value cell. When a variable is rebound, a *name/old-value* pair is stored on the stack and its new value is placed in the value cell. At the time of unbinding, the old value must be restored. During a spaghetti context switch between two environments the values of all variables not common to both environments have to be restored. Shallow binding eventually was chosen over deep binding to improve performance by eliminating the stack search required in deep binding; however, the only major improvement appears to be an increase in the speed of the Interlisp-10 Interpreter [21].

We deliberately chose deep over shallow binding in Interlisp-VAX. There are substantial tradeoffs between deep and shallow binding for spaghetti stacks (multiple stack groups). With Interlisp-10, the expense of shallow bindings in spaghetti manipulations results in spaghetti stacks that are perceived as too slow for use in many applications.

The basic performance tradeoffs are these: For shallow binding, the basic bind/unbind operations are marginally more expensive; locality of reference is less; "free variable" lookup is free; and spaghetti stack environment switching is expensive, because it involves rebinding all variables on both threads back to a common root. For deep binding, free variable lookup is expensive, but spaghetti operations are cheap. For a system without spaghetti, the performance tradeoff is clearly in favor of shallow binding. For a system that actually uses spaghetti the tradeoff is less clear.

The choice of deep over shallow binding has proven to be even more critical than at first realized. The original deep-binding scheme was straightforward; free variable lookup occurred at every reference. Preliminary timings showed that 15 to 40 percent of all time was spent in the free variable lookup routine and that the interpreter was slow. A simple caching scheme to reduce the frequency of scans has been implemented and more elaborate schemes are under consideration. At present the overhead to support deep binding is typically the largest single item in the performance of average programs. In retrospect, this one choice is the most obviously questionable we have made. We at least should have considered performance optimizations for deep binding *from the beginning*, and possibly should have chosen shallow binding.

5.3 Integer Representation

The representation of integers is one of the more unusual aspects of Interlisp-VAX. Unlike most LISPs implemented on machines without tagged architectures, Interlisp-VAX has no notion of integer number boxes. The VAX architecture has a 32-bit address space, where the high-order bit selects system space or user space. Since user programs cannot have addresses in the range 2↑31 to 2↑32, these *addresses* are used to provide 2↑31 immediate integers. The loss of 1 bit of integer precision is not significant to Interlisp applications, which typically have not used much arithmetic. The gain is considerable as all integer arithmetic is fast.

5.4 Other Objects

All other objects are organized by two parameters. All are treated similarly by memory management and garbage collection. They have similar data fetch and store procedures. Our preference here was to keep the representation as simple as possible and still retain reasonable implementation efficiency. Two methods allow a few simple data primitives to describe all possible operations on Interlisp-VAX's data.

All objects are described by a single, separate type descriptor, which includes the length of the objects and the number of pointers they contain. The pointers are always allocated at the beginning of the object. All user-defined types and all basic data types except arrays and strings are in this group.

All variable-length objects are described by a *sequence descriptor*. These include arrays, code arrays, and strings. There is only one type of sequence descriptor. All data access to objects described by sequence descriptors takes exactly the same form (e.g., fetching the nth character of a string involves exactly the same arithmetic as fetching the nth item of an array).

6. Current Status

A compatible Interlisp running on the VAX is presently available. Several large Interlisp systems have been transported onto the VAX with little difficulty. Some of the systems now running on the VAX are Affirm [8, 22] (a program verification system), AP3 (an AI programming language), Hearsay-III [7] (a domain-independent framework for building knowledge-based expert systems), Consul [4] (a knowledge-based user interface to interactive tools) and KL-One [13] (a language for representing knowledge as a semantic network). The current speed is about one-third of a DEC-KL. The system is being tuned to improve the performance. Our goal is to reach the speed of approximately one-half of a DEC-KL.

7. The Difficulties of Implementing Interlisp

Implementing Interlisp is a more difficult task than implementing "any LISP," as the task is strongly constrained. Generally speaking, it is not acceptable to heavily modify Interlisp programs in order to make them run on another machine. Although it is not difficult to create a LISP that resembles Interlisp, the result is not Interlisp and will not support the multitude of code already written in Interlisp. Particularly, the Interlisp environment [20, 18] is essential as part of a recognizable Interlisp system, and the environment is dependent on all the quirks and hidden features of the original Interlisp-10.

The VM specification is an invaluable aid in producing a new Interlisp. However, its relatively small size (126 pages) makes the task of implementing Interlisp seem deceptively easy. If only the functions of the VM were implemented, the result would not be Interlisp. There are many small but important features that are not mentioned in the VM specification, the Interlisp manual, nor anywhere else. Many nonobvious constraints remain to be discovered by a potential implementor. Subtle interactions of seemingly straightforward features must be anticipated or they will be discovered as bugs.

To achieve reasonable efficiency with Interlisp, conceptually simple data structures can require complex representations. For example, great care has been taken to use the correct VAX instruction necessary for a function call (*CALLS* instruction) in order to attain an acceptable speed. If it were not necessary to be concerned with the spaghetti stack for the convenience of the VAX, the task would be easier, but again Interlisp would not result. As complexity increases, planning, implementing, debugging, and maintenance become more difficult.

8. Conversion Problems Encountered by Users

Users face three classes of problems in converting large Interlisp programs to run on the VAX (most programs now originate from Interlisp-10). The first class of problems arises because Interlisp-10 is a shallow-bound system while Interlisp-VAX is a deep-bound system. Programs running on a deep-bound system must be more selective with compiler declarations than programs running on a shallow-bound system (otherwise the programs might not run). Shallow-bound systems are less restrictive in terms of compiler declarations and coding style. This difference creates problems because certain compiler declarations that are possible in a shallow-bound system will not work in a deep-bound system. Specifically, in a deep-bound system the variables bounded by RESETVARS must be GLOBALVAR, and the variables bounded by PROG cannot be GLOBALVAR. For example, the following program will compile only in a shallow-bound system:

```
(FOO
  [LAMBDA (X)
    (DECLARE: (SPECVARS Y)
              (GLOBALVARS HELPFLG))
    (PROG (HELPFLG Y I)
          (... ])
```

To make the program compile in a deep-bound system it must be re-coded to:

```
(FOO
  [LAMBDA (X)
    (DECLARE: (SPECVARS Y)
              (GLOBALVARS HELPFLG))
    (RESETVARS (HELPFLG)
               (RETURN (PROG (Y I)
                             (... ])
```

The second class of problems arises because the program is running under a different operating system with different restrictions. Although it is obvious that any direct call to a JSYS (a TENEX or TOPS-20 operating system call) must be removed, other problems can arise. The greatest limitation of UNIX is that file names are restricted to fourteen characters and have no version numbers.

The third problem comes closest to being the Achilles' heel of the implementation, and illustrates the importance of considering all of the implications of representation decisions. Because of the representation of integers in Interlisp-VAX, taking CAR of a number generates a machine check. A surprising number of programs do this without the programmer being aware of the action, canonically:

```
(AND (EQ (CAR X) (QUOTE QUOTE) --)
```

where X has not been guaranteed to be a LISTP. In the previous implementations, CAR and CDR of numbers and other non-lists were harmless, if meaningless, operations.

9. Remaining Tasks

Interlisp-VAX achieves compatibility with Interlisp-10 and Interlisp-D. However, continuous effort is necessary to maintain compatibility with still-evolving Interlisp-D (currently Interlisp-10 is very stable and is expected to remain so). A few functions remain to be coded, and of course there are still bugs to remove. A major concern is to increase the speed of Interlisp-VAX. Currently the compiler is being changed to produce better code for the VAX (by using peephole optimization and better register allocation, and producing more compact code). A VMS [23] version using EUNICE should be available by the time this paper is published.

We are considering more optimizations of free variable lookup, and may eventually experiment with shallow binding. Some minor implementation choices may have unexpectedly large impact on the total performance, for example, the fact that atom PNAMEs are really strings, including a superfluous string header, so inspecting the PNAME involves touching *three* different areas of memory. There is much more "open coding" to be done. Currently, only the primitive predicates and some integer arithmetic are compiled in line, with the result that Interlisp-VAX code still is top heavy with function calls. Finally, there is much more to be done in cooperation with improvements in UNIX to reduce the size of SYSOUT and increase the effective sharing of active pages among copies from zero to some reasonable level.

10. Conclusions

The implementation of Interlisp on the VAX is a technical success, and should prove to be a popular and useful tool within the combined VAX and Interlisp communities. Among the most important contributing elements to our success were the reliance on proven software technology, careful planning in advance of coding, the availability of reliable specifications for what was to be produced, and especially the ability to share large masses of code with other Interlisp implementations.

11. Acknowledgments

This project would have been immeasurably more difficult and less likely to succeed without the cooperation of Xerox Corporation: historical contributions to Interlisp-10, support for the writing of the Interlisp Virtual Machine Specification, sharing of code developed for Interlisp-D, and the personal assistance and advice of the staff of the Xerox Palo Alto Research Center.

The following members of the ISI staff have contributed to the implementation of the Interlisp-VAX: Dan Lynch, Bob Balzer, Andrea Ignatowski, Steve Saunders, and Don Voreck.

References

1. *BLISS-10 Programmer's Reference Manual*, Digital Equipment Corporation, Maynard, Mass., 1974.

2. Bobrow, D. G., J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, "TENEX, a paged time sharing system for the PDP-10," *Communications of the ACM* 15, (3), March 1972, 135-143.

3. Bobrow, D. G., and B. Wegbreit, "A model and stack implementation for multiple environments," *Communications of the ACM* 16, (10), October 1973, 591-603.

4. Brachman, R., *A Structural Paradigm for Representing Knowledge*, Bolt, Beranek, and Newman, Inc., Technical Report, 1978.

5. Burton, R. R., et al., "Overview and status of DoradoLISP," in *Proceedings of the 1980 LISP Conference*, pp. 243-247, Stanford, Calif., August 1980.

6. Cheney, C. J., "A Non-recursive list compacting algorithm," *Communications of the ACM* 13, (11), November 1970, 677-678.

7. Erman, L., P. London, and S. Fickas, "The design and an example use of Hearsay-III," in *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pp. 409-415, Vancouver, B.C., August 1981.

8. Gerhart, S. L., et al., "An overview of Affirm: A specification and verification system," in *Proceedings IFIP 80*, pp. 343-348, Australia, October 1980.

9. Jensen, K., and N. Wirth, *Pascal User Manual and Report*, Springer-Verlag, New York, 1975.

10. Kernighan, B. W., and D. M. Richie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, N. J., 1978.

11. Knight, T., D. Moon, J. Holloway, and G. Steele, *CADR*, Massachusetts Institute of Technology, Technical Report 528, June 1979.

12. Koomen, J. A.G.M., The Interlisp Virtual Machine: A Study of its Design and its Implementation as Multilisp, Master's thesis, University of British Columbia, 1980.

13. Mark, W., "Representation and inference in the Consul System," in *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pp. 375-381, Vancouver, B.C., August 1981.

14. Moon, D. A., *Maclisp Reference Manual*, Massachusetts Institute of Technology, Laboratory for Computer Science, Technical Report, March 1974.

15. Moore, J S., *The Interlisp Virtual Machine Specification*, Xerox Palo Alto Research Center, Technical Report CSL 76-5, March 1979.

16. Richards, M., and C. Whitby-Strevens, *BCPL - The Language and its Compiler*, Cambridge University Press, New York, 1979.

17. Ritchie, D. M., and K. Tompson, "The UNIX time-sharing system," *Communications of the ACM* 17, (7), July 1974, 365-375.

18. Sandewall, E., "Programming in an interactive environment: The LISP experience," *Computing Surveys* 10, (1), March 1978, 35-71.

19. Teitelman, W., *Interlisp Reference Manual*, Palo Alto, Calif., 1978.

20. Teitelman, W., and L. Masinter, " The Interlisp programming environment," *Computer* 14, (4), April 1981, 25-33.

21. Teitelman, W., Shallow bindings in Interlisp-10 (April 22, 1976). Note to Interlisp Users, Xerox Memo.

22. Thompson, D. H., S. L. Gerhart, R. W. Erickson, S. Lee, and R. L. Bates (eds.), *The Affirm Reference Library*, USC/Information Sciences Institute, 1981. (Five volumes: Reference Manual, User's Guide, Type Library, Annotated Transcripts, and Collected Papers; 450 pages.)

23. *VAX Software Handbook*, Digital Equipment Corporation, 1980.

24. *VAX Architecture Handbook*, Digital Equipment Corporation, 1981.

25. Weinreb, D., and D. Moon, *LISP Machine Manual*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Technical Report, January 1979.

26. White, J. L., "NIL - A perspective," in *Macsyma Users' Conference Proceedings*, June 1979.

27. White, J. L., "Address/memory management for a giganitic LISP environment or, GC considered harmful," in *Proceedings of the 1980 LISP Conference*, pp. 119-127, Stanford, Calif., August 1980.