Palo Alto Research Center

Papers on Interlisp-D

B. A. Sheil and Larry M. Masinter, editors

XEROX

Papers on Interlisp-D

B. A. Sheil and Larry M. Masinter, editors

September 1980; Revised July 1981; Revised January 1983

Cognitive and Instructional Sciences Series
CIS-5

Corporate Accession P83-00029

XEROX

PALO ALTO RESEARCH CENTERS
3333 Coyote Hill Road / Palo Alto / California 94304

INTRODUCTION

Interlisp-D is an implementation of the Interlisp programming system [Teitelman *et al.*, 78] for the Xerox D-series of personal computers. It is a complete, upward compatible extension of the original implementation of Interlisp for the DEC PDP-10 (hereafter, Interlisp-10) which supports all of the Interlisp system software and all but the explicitly machine dependent application programs which were developed for Interlisp-10.

The principal areas of extension reflect the fact that Interlisp-D operates in a personal machine computing environment. Thus, many new facilities have been added to take advantage of the interactive graphics and communication facilities available on these machines [XEOS, 82]. As a result (and also in the interests of future transportability), many facilities that previous implementations have obtained from their host machine operating systems have been implemented within Lisp as part of Interlisp-D. Thus, Interlisp-D is also, in effect, a dramatic extension "downwards" of the Interlisp Virtual Machine Specification [Moore, 76]. Unfortunately, there does not yet exist a formal document like [Moore, 76] describing this simpler, more environment independent, kernel.

This report contains five papers which motivate and describe the Interlisp-D system. It is the third (and a major) revision of the original *Papers on Interlisp-D* report. As the Interlisp-D system has become more widely used and formally supported, two of the original papers (*The Interlisp-D I/O system* and *The Interlisp-D display facilities*) have been superceded by documentation [XEOS, 82, 83]. In their place, two papers have been included on the historical development of Interlisp and the style of exploratory programming that it is designed to support. Of the other three papers, which appeared in the previous edition of this report, two were presented at the *1980 Lisp Conference* and are reprinted here, with slight changes, so as to make them more widely available. The other, which appeared in *SIGART Newsletter*, provides a more recent report on the system's status and probable future development. The papers are:

Environments for exploratory programming (Jan 83)
Describes and motivates the style of programming which the Interlisp environment was designed to support.

The Interlisp programming environment (Nov 80)
Describes the common Interlisp environment, its history as of late 1980, and some of the reasons why it developed the way it did.

Interlisp-D: Overview and status (Jun 80)
Describes the Interlisp-D implementation, its goals and techniques, and our subsequent reflections thereon.

Interlisp-D: Further steps in the flight from time-sharing (Jun 81)
A status report and description of ongoing and planned extensions, as of June 1981.

Local optimization in a compiler for stack based Lisp machines (Jun 80)
Describes the optimizations used during compilation of Lisp into the special purpose Lisp instruction set and their observed effectiveness.

As documented in the *Overview and status* paper, the implementation of Interlisp-D has been a major effort, both in terms of the time taken and the number of people who have contributed. The integration of Interlisp's programming support tools into the personal computing environment has been and continues to be a design task of similar magnitude. Our hope is that these papers provide both a clear description of its motivation, a snapshot of its current state, and some useful ideas for other implementors of integrated programming environments, both for Lisp and for other languages.

REFERENCES

Moore, J.
The Interlisp virtual machine specification. Xerox PARC, CSL-76-5, 1976.

Teitelman, W. *et al.*
Interlisp reference manual. Xerox PARC, 1978.

XEOS
Interlisp-D Users Guide. Xerox Electo-Optical Systems, Pasadena, California, September 1982.

Interlisp reference manual. XEOS, 1983 (forthcoming).

Environments for exploratory programming

B. A. Sheil

An oil company needs a system to monitor and control the increasingly complex and frequently changing equipment used to operate an oil well. A electronic circuit designer plans to augment a circuit layout program to incorporate a variety of vaguely stated "design rules". A newspaper wants a page layout system to assist editors in balancing the interlocking constraints that govern the placement of stories and advertisments. A government agency envisions a personal workstation that would provide a single, integrated interface to a variety of large, evolving data base systems.

Applications like these are forcing the commercial deployment of a radically new kind of programming system. First developed to support research in artificial intelligence and interactive graphics, these new tools and techniques are based on the notion of *exploratory programming*, the conscious intertwining of system design and implementation. Fueled by dramatic changes in the cost of computing, such *exploratory programming environments* have become, virtually overnight, a commercial reality. No fewer than four such systems were displayed at NCC '82 and their numbers are likely to increase rapidly as their power and range of application become more widely appreciated.

EXPLORATION AND IMPLEMENTATION

Despite the diversity of subject matter, a common thread runs through our example applications. They are, of course, all large, complex programs whose implementations will require significant resources. Their more interesting similarity, however, is that it is extremely difficult to give complete specifications for any of them. The reasons range from sheer complexity (the circuit designer can't anticipate all the ways in which all the potential design rules might interact), through continually changing requirements (the equipment in the oil rig changes, as do the information bases that the government department is required to consult), to the subtle human factors issues which determine the effectiveness of an interactive graphics interface.

Whatever the cause, a large programming project with uncertain or changing specifications is a particularly deadly combination for conventional programming techniques. Virtually all of modern programming methodology is predicated on the assumption that a programming project is fundamentally a problem of *implementation*, rather than *design*. The design is supposed to be decided on *first*, based on specifications provided by the client; the implementation follows. This dichotomy is so important that it is standard practice to recognize that a client may have only a partial understanding of his needs, so that extensive consultations may be required to ensure a complete specification with which the client will remain happy. This dialogue ensures a fixed specification which will form a stable base for an implementation.

The vast bulk of existing programming practice and technology, such as structured design methodology, is designed to ensure that the implementation does, in fact, follow the

---

A version of this paper is to appear in Datamation, *February 1983.*

specification in a controlled fashion, rather than wander off in some unpredictable direction. And for good reason. Modern programming methodology is a significant achievement that has played a major role in preventing the kind of implementation disasters that often befell large programming projects in the 1960s.

The implementation disasters of the 1960s, however, are slowly being succeeded by the design disasters of the 1980s. The projects described above simply will not yield to conventional methods. Any attempt to obtain an exact specification from the client is bound to fail because, as we have seen, the client *does not know and cannot anticipate* exactly what is required. Indeed, the most striking thing about these examples is that the clients' statements of their problems are really *aspirations*, rather than *specifications*. And since the client has no experience in which to ground these aspirations, it is only by exploring the properties of some putative solutions that the client will find out what is really needed. No amount of interrogation of the client or paper exercises will answer these questions; one just has to try some designs to see what works.

The consequences of approaching problems like these as routine implementation exercises are dramatic. First, the implementation team begins by pushing for an exact specification. How long the client resists this coercion depends on how well he really understands the limits of his own understanding of the problem. Sooner or later, however, with more or less ill-feeling, the client accepts a specification and the implementation team goes to work. The implementors take the specification, partition it, define a module structure that reflects this partitioning, freeze the interfaces between them, and repeat this process until the problem has been divided into a large number of small, easily understood and easily implementable pieces. *Control* over the implementation process is achieved by the imposition of *structure* which is then *enforced* by a variety of management practices and programming tools.

Since the specification, and thus the module structuring, is considered fixed, one of the most effective methods for enforcing it is the use of *redundant descriptions* and *consistency checking*. Thus the importance of techniques such as interface descriptions and static type checking, which require multiple statements of various aspects of the design in the program text in order to allow mechanical consistency checks to ensure that each piece of the system remains consistent with the rest. In a well executed conventional implementation project, a great deal of internal rigidity is built into the system in this way in the interests of ensuring its orderly development.

The problems emerge, usually at system acceptance time, when the client requests, not just superficial, but *radical* changes, either as a result of examining the system, or for some completely exogenous reason. From the point of view of conventional programming practice, this indicates a failure at specification time. The software engineer should have been more persistent in obtaining a fuller description of the problem, in involving all the effected parties, *etc.*. And this is often true. Many ordinary implementation exercises are brought to ruin by insufficient attention having been paid to getting the consequences of the specification fully agreed to. But that's not the problem here. The oil company didn't know about the new piece of equipment whose behavior is very different from the existing equipment on which the specification was based. No one knew that the layout editors would complain that it doesn't "feel right" now that they can no longer physically handle the copy (even in retrospect, it's unclear why they feel that way and what to do about it). *Etc., etc.,*

etc.. Neither would any amount of speculation by either client or software engineer have helped. Rather, it would just have prompted an already nervous client to demand whole dimensions of flexibility that would *not* in fact be needed, leaving the system just as unprepared for the ones that eventually turned out to matter.

Whatever the cause, the implementation team has to rework the system to satisfy a new, and significantly different, specification. That puts them in a situation that conventional programming methodology simply refuses to acknowledge (except as something to avoid). As a result, their programming tools and methods are suddenly of limited effectiveness, if not actually counterproductive. The redundant descriptions and imposed structure that were so effective in constraining the program to follow the old specification have lost none of their efficacy - they *still* constrain the program to follow the old specification. And they're difficult to change. The whole point of redundancy is to protect the design from a single (unintentional) change. But it's equally well protected against a single *intentional* change. Thus, all the changes have to be made all over the place. (And, since this should never happen, there's no methodology to guide or programming tools to assist this process.) Of course, if the change is small (as it "should" be), there is no particular problem. But if it is large, so that it cuts across the module structure, the implementation team finds that they literally have to fight their way out of their previous design.

Still no major problem, if that's the end of the matter. But it rarely is. The new system will suggest yet another change. And so on. And on. After a few iterations of this, not only are the client and the implementation team not on speaking terms, but the repeated assaults on the module structure have likely left it looking like spaghetti. It still gets in the way (firewalls are just as impenetrable if laid out at random as they are when laid out straight), but has long ceased to be of any use to anyone except to remind them of the sorry history. Increasingly, it is actively subverted (enter LOOPHOLES, UNSPECs, *etc.*) by programmers whose patience is running thin. Even were the design suddenly to stabilize (unlikely in the present atmosphere), all the seeds have now been sown for an implementation disaster as well.

PROGRAMMING AS EXPLORATION

The alternative to this kind of predictable disaster is *not* to abandon structured design for programming projects which are, or which can be made, well-defined. That would be a tremendous step backwards. Instead, we should recognize that some applications are best thought of as *design problems*, rather than implementation projects. These problems require programming systems which allow the design to emerge from experimentation with the program, so that design and program develop together. Environments in which this is possible were first developed in artificial intelligence and computer graphics, two research areas which are particularly prone to specification instability.

At first sight, artificial intelligence might seem to be an unlikely source of programming methodology. However, constructing programs, in particular, programs which carry out some intelligent activity, is central to artificial intelligence. Since almost any intelligent activity is likely to be poorly understood (once something becomes well understood we usually no longer consider it "intelligent"), the artificial intelligence programmer invariably has to restructure his program many, many times before it becomes reasonably proficient. In addition, since intelligent activities are complex, the programs tend to be very large, yet they

Xerox 1108 Interlisp-D system

An exploratory programming system designed to be installed in the user's office. Processor, main memory (1.5 MBytes), rigid and flexible local disks, and Ethernet connection are all contained in the processor cabinet at lower right. The "mouse" pointing device, which moves a cursor image over the display according to sensed horizontal motion across the table, can be seen to the right of the keyboard, in front of the display.

Photo: Ken Beckman

are invariably built by very small teams (often a single researcher). Consequently, they are usually at or beyond the manageable limits of complexity for their implementors. In response, a variety of programming environments based on the Lisp programming language have evolved to aid in the development of these large, rapidly changing systems.

The rapidly developing area of interactive graphics has encountered similar problems. Fueled by the rapid drop in the cost of computers capable of supporting interactive graphics, there has been an equally rapid development of applications which make heavy use of interactive graphics in their user interfaces. Not only was the design of such interfaces almost completely virgin territory as little as ten years ago, but even now, when there are a variety of known techniques (*e.g.*, menus, windows, *etc.*) for exploiting this power, it is still very difficult to determine how easy it will be to use a proposed user interface and how well it will match the user's needs and expectations in particular situations. Consequently, complex interactive interfaces usually require extensive empirical testing to determine whether they are really effective and considerable redesign to make them so. While interface design has always required some amount of tuning, the vastly increased range of possibilities available in a full graphics system has made the design space unmanageably large to explore without extensive experimentation. In response, a variety of systems, of which Smalltalk is the most well known, have been developed to facilitate this experimentation by providing a wide range of built in graphical abstractions and methods of modifying and combining them together into new forms.

EXPLORATORY PROGRAMMING SYSTEMS

In contrast to conventional programming technology, which *restrains* the programmer in the interests of orderly development, exploratory programming systems must *amplify* the programmer in the interests of maximizing his effectiveness. Exploration in the realm of programming can require small numbers of programmers to make essentially arbitrary transformations to very large amounts of code. Such programmers need *programming power tools* of considerable capacity or they will simply be buried in detail. So, like an amplifier, their programming system must magnify their necessarily limited energy and minimize extraneous activities that would otherwise compete for their attention.

One source of such power is the use of interactive graphics. Exploratory programming systems have capitalized on recent developments in personal computing with extraordinary speed. Consider, for example, the Xerox 1108 Interlisp-D system shown on the facing page. The large format display and "mouse" pointing device allow very high bandwidth communication with the user. Exploratory programming environments have been quick to seize on the power of this combination to provide novel programming tools, as we shall see.

In addition to programming tools, these personal machine environments allow the standard features of a professional workstation, such as text editing, file management and electronic mail, to be provided *within the programming environment itself*. Not only are these facilities just as effective in enhancing the productivity of programmers as they are for other professionals, but their integration into the programming environment allows them to be used at any time during programming. Thus, a programmer who has encountered a bug can send a message reporting it while remaining within the debugger, perhaps including in the message some information, like a backtrace, obtained from the dynamic context.

Another apparent source of power is to build the important abstract operations and objects of some given application area directly into the exploratory environment. All programming systems do this to a certain extent; some have remarkably rich structures for certain domains, for example, the graphics abstractions embedded within Smalltalk. If the abstractions are well chosen, this approach can yield a powerful environment for exploration within the chosen area, because the programmer can operate entirely in substantively meaningful abstractions, taking advantage of the considerable amount of implementation and design effort that they represent.

The limitations of this approach, however, are clear. Substantive abstractions are necessarily effective only within a particular topic area. Even for a given area, there is generally more than one productive way to partition it. Embedding one set of abstractions into the programming system encourages developments that "fit" within that view of the world at the expense of others. Further, if one enlarges one's area of activity even slightly, a set of abstractions that was once very effective may become much less so. In that situation, unless there are effective mechanisms for reshaping the built in abstractions to suit the changed domain, users are apt to persist with them, at the cost of distorting their programs. Embedded abstractions, useful though they are, by themselves enable only *exploration in the small*, confined within the safe borders where the abstractions are known to be effective. For *exploration in the large*, a more general source of programming power is needed.

Of course, the exact mechanisms which different exploratory systems propose as essential sources of programming power vary widely, and these differences are hotly debated within their respective communities. Nevertheless, despite strong surface differences, the systems share some unusual characteristics at both the language and environment level.

Languages

The key property of the programming languages used in exploratory programming systems is their emphasis on *minimizing and deferring the constraints* placed on the programmer, in the interests of minimizing and deferring the cost of making large scale program changes. Thus, not only are the conventional structuring mechanisms based on redundancy not used, but the languages make extensive use of *late binding*, *i.e.*, allowing the programmer to defer commitments as long as possible.

The clearest example is that exploratory environments invariably provide dynamic storage allocation with automatic reclamation (garbage collection). To do otherwise imposes an intolerable burden on the programmer to keep track of all the paths through his program that might access a particular piece of storage to ensure that none of them access or release it prematurely (and that someone does release it eventually!). This can only be done either by careful isolation of storage management or with considerable administrative effort. Both are incompatible with rapid, unplanned development, so neither is acceptable. Storage management must be provided by the environment itself.

Other examples of late binding include the *dynamic typing* of variables (associating data type information with a variable at run-time, rather than in the program text) and the *dynamic binding* of procedures. The freedom to defer deciding the *type* of a value until run-time is important because it allows the programmer to experiment with the type structure itself.

Usually, the first few drafts of an exploratory program implement most data structures in general, inefficient structures such as linked lists, discriminated (when necessary) on the basis of their contents. As experience with the application evolves, the critical distinctions which determine the type structure are themselves determined by experimentation, and may be among the last, rather than the first, decisions to evolve. Dynamic typing makes it easy for the programmer to write code which keeps these decisions as tacit as possible.

The dynamic binding of procedures is more than a simple load-time linkage. It allows the programmer to change dynamically the subprocedures invoked by a given piece of code, simply by changing the run-time context. The simplest form of this is to allow procedures to be used as arguments or as the value of variables. More sophisticated mechanisms allow procedure values to be computed or even encapsulated inside the data values on which they are to operate. This packaging of data and procedures into a single object, known as "object oriented programming", is a very powerful technique. For example, it provides an elegant, modular solution to the problem of generic procedures (i.e., every data object can be thought of as providing its own definition for common actions, such as printing, which can be invoked in a standard way by other procedures). For these reasons, object oriented programming is a widely used technique in exploratory programming, and actually forms the basic programming construct of the Smalltalk language.

The dynamic binding of procedures can be taken one step further when procedures are represented as data structures which can be effectively manipulated by other programs. While this is of course possible to a limited extent by reading and writing the text of program source files, it is of much greater significance in systems that define an explicit representation for programs as syntax trees or their equivalent. This, coupled with the interpreter or incremental compiler provided by most exploratory programming systems, is an extraordinarily powerful tool. Its most dramatic application is in programs that construct other programs that they later invoke. This technique is often used in artificial intelligence in situations where the range of possible behaviors is too large to encode efficiently as data structures but can easily be expressed as combinations of procedure fragments. An example might be a system which "understands" instructions given in natural language by analyzing each input as it is received, building a *program* which captures its "meaning", and then *evaluating* that program to achieve the requested effect.

Aside from such specialized applications, effective methods for mechanically manipulating procedures enable two other significant developments. The first is the technique of program development by writing interpreters for special purpose languages. Once again, this is a basic technique of artificial intelligence that has much wider applicability. The key idea is that one develops an application by designing a special language in which the application is relatively easy to state. Like any notation, such a language provides a concise representation which suppresses common or uninteresting features in favor of whatever the designer decides is more important. A simple example is the use of notations like context free grammars (BNF) to "meta-program" the parsers for programming languages. Similar techniques can be used to describe, among other things, user interfaces, transaction sequences, and data transformations. Application development in this framework is a dialectic process of designing the application language and developing an interpreter for it, since both the language and the interpreter will evolve during development. The simplest way of doing this is to evolve the application language out of the base provided by the

development language. Simply by allowing the application language interpreter to call the development language interpreter, expressions from the development language can be used wherever the application language currently has insufficient power. As one's understanding of the problem develops, the application language becomes increasingly powerful and the need to escape into the development language becomes less important.

Programming tools

The second result of having procedures be easily manipulated by other procedures is that it becomes easy to write program manipulation subsystems. This in turn has two key consequences. First, the exploratory programming language itself can grow. The remarkable longevity of Lisp in the artificial intelligence community is in large part due to the language having been repeatedly extended to include modern programming language syntax and constructions. The vast majority of these extensions were accomplished by defining source to source transformations which converted the new constructions into more conventional Lisp expressions. The ease with which this can be done allows each user, and even each project, to extend the language to capture the idioms that are found to be locally useful.

Second, the accessibility of procedures to mechanical manipulation facilitates the development of programming support tools. All exploratory programming environments boast a dazzling profusion of programming tools. To some extent, this is a virtue of necessity, as the flexibility necessary for exploration has been gained at considerable sacrifice in the ability to impose structure. That loss of structure could easily result in a comensurate loss of control by the programmer. The programming tools of the exploratory programming environment enable the programmer to reimpose the control that would be provided by structure in conventional practice.

Programming tools achieve their effectiveness in two quite different ways. Some tools are simply effective *viewers* into the user's program and its state. Such tools permit one to find information quickly, display it effectively, and modify it easily. A wide variety of tools of this form can be seen in the two Interlisp-D screen images on page 11, including data value *inspectors* (which allow a user to look at and modify the internal structure of an object), *editors* for code and data objects, and a variety of *break and tracing* packages. Especially when coupled with a high bandwidth display, such viewers are very effective programming tools.

The other type of programming tool is *knowledge based*. Viewer based tools, such as a program text editor, can operate effectively with a very limited "understanding" of the material with which they deal. By contrast, knowledge based tools must know a significant amount about the *content* of a user's program and the *context* in which it operates. Even a very shallow analysis of a set of programs (*e.g.*, which programs call which other ones) can support a variety of effective programming tools. A *program browser* allows a programmer to track the various dependencies between different parts of a program by presenting easy to read summaries which can be further expanded interactively. Deeper analysis allows more sophisticated facilities. The Interlisp program analyser (Masterscope) has a sufficiently detailed knowledge of Lisp programs that it can provide a complete static analysis of an arbitrary Lisp program. A wide variety of tools have been constructed which use the

database provided by this analysis to answer complex queries (which may require significant reasoning, such as computing the transitive closure of some property), to make systematic changes under program control (such as making some transformation wherever a specified set of properties hold), or to check for a variety of inconsistent usage errors.

Finally, integrated tools provide yet another level of power. The Interlisp system "notices" whenever a program fragment is changed (by the editor, or by redefinition). The program analyser is then informed that any existing analysis is invalid, so that incorrect answers are not given on the basis of old information. The same mechanism is used to notify the program management subsystem (and eventually the user, at session end) that the corresponding file needs to be updated. In addition, the system will remember the previous state of the program, so that at any subsequent time the programmer can undo the change and retreat (in which case, of course, all the dependent changes and notifications will also be undone). This level of cooperation between tools not only provides immense power to the programmer, but it relieves him of a level of detail that he would otherwise have to manage himself. The result is that more attention can be paid to exploring the design.

Contraction

A key, but often neglected, component of an exploratory programming system is a set of facilities for program *contraction*. The development of a true exploratory program is "design limited", so that is where the effort has to go. Consequently, the program is often both inefficient and inelegant when it first achieves functional acceptability. If the exploration is an end in itself, this might be of limited concern. However, it is more often the case that a program developed in an exploratory fashion must eventually be used in some real situation. Sometimes, the time required to reimplement (using the prototype program as a specification) is prohibitive. Other times, the choice of an exploratory system was made to allow for expected *future* upheaval, so it is essential to preserve design flexibility. In either event, it is necessary to be able to take the functionally adequate program and transform it into a program whose efficiency is comparable to the best program one could have written, in any language, had only one known what one was doing when one started.

The importance of being able to make this *post hoc* optimization cannot be overemphasized. Without it, one's exploratory programs will always be considered "toys"; the pressure to abandon the exploratory environment and start implementing in a "real" one will be overwhelming; and, once that move is made (and it is *always* made too soon), exploration will come to an end. The requirement for efficient implementation places two burdens on an exploratory programming system. First, the architecture has to *permit* efficient implementations. Thus, for example, the obligatory automatic storage manager must either be so efficient that it imposes negligible overhead, or it must allow the user to circumvent it (*e.g.*, to allocate storage statically) when and where the design has stabilized enough to make this optimization possible.

Second, as the performance engineering of a large system is almost as difficult as its initial construction, the environment must provide *performance engineering tools*, just like it provides design tools. These include good instrumentation, a first class optimizing compiler, program manipulation tools (including, at the very least, full functionality compiler macros), and the ability to add declarative information where necessary to guide the program

[Continued on page 12]

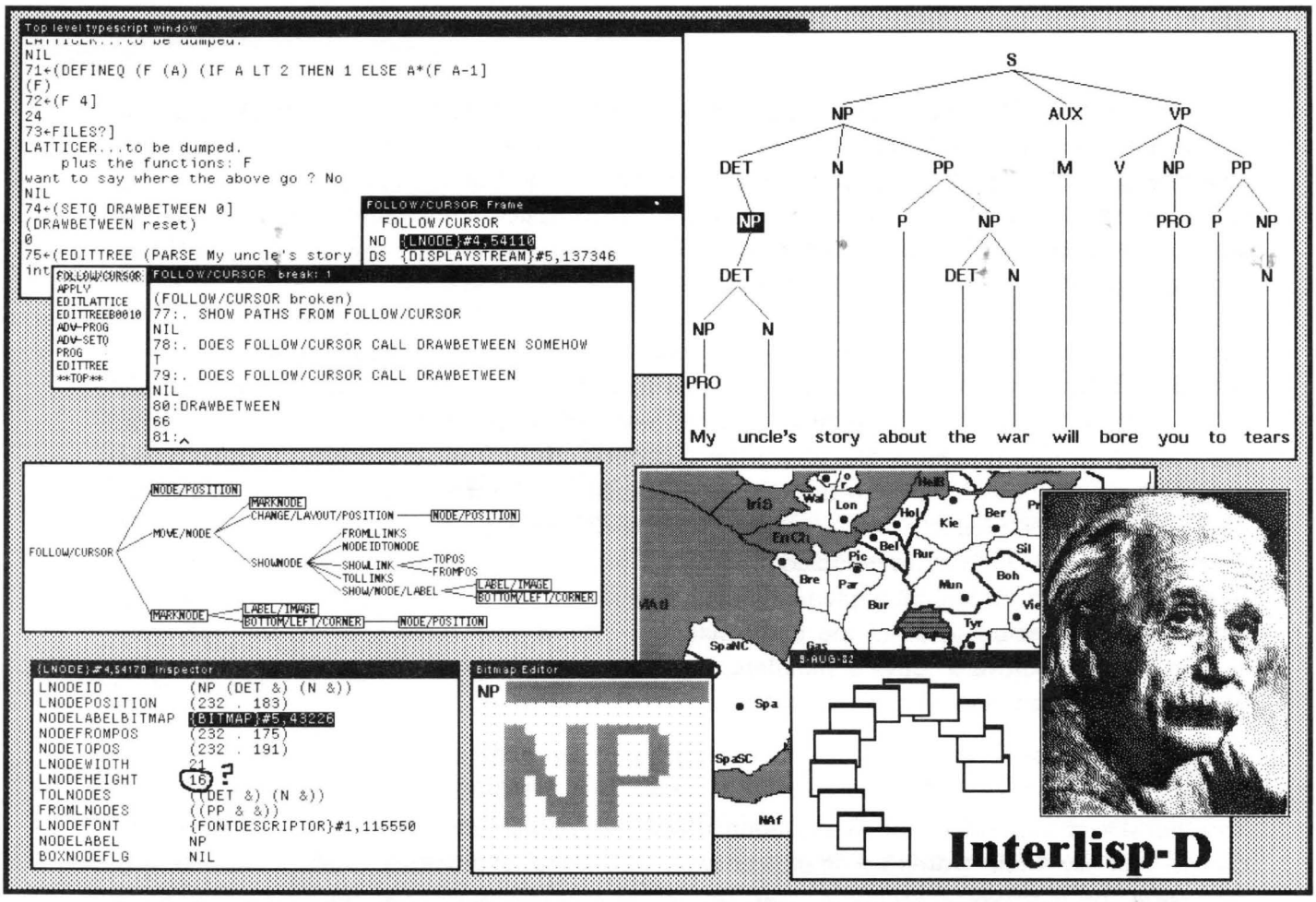
The two screen images at right show some of the exploratory programming tools provided in the Interlisp-D environment. The screen is divided into several rectangular areas or windows, *each of which provides a view onto some data or process and which can be reshaped and repositioned at will by the user. When they overlap, the occluded portion of the lower window is automatically saved, so that it can be restored when the overlapping window is removed. Since the display is bitmapped, each window can contain an arbitrary mixture of text, lines, curves, and half-tone and solid area images.*

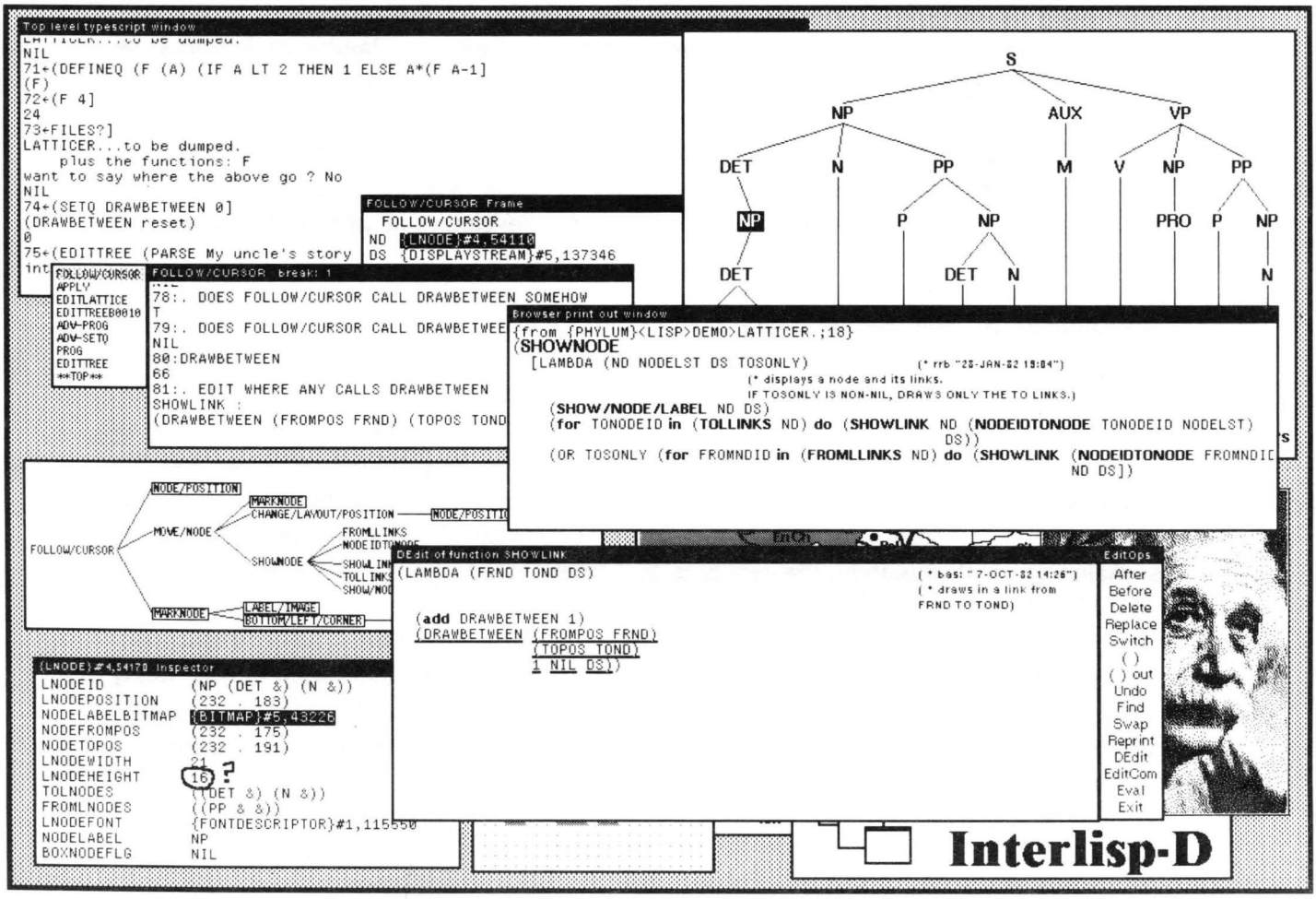
In the typescript window (upper left), the user has defined a program F (factorial) and has then immediately run it, giving an input of 4 and getting a result of 24. Next, he queries the state of his files, finding that one file has been changed (previously) and one function (F) has been defined but not associated with any file yet. The user sets the value of DRAWBETWEEN *to 0 in command 74, and the system notes that this is a change and adds* DRAWBETWEEN *to the set of "changed objects" that might need to be saved.*

Then, the user runs the program EDITTREE, *giving it a parse tree for the sentence "My uncle's story about the war will bore you to tears". This opens up the big window on the right in which the sentence diagram is drawn. Using the mouse, the user starts to move the* NP *node on the left (which is inverted to show that it is being moved). While the move is taking place, the user interrupts the tree editor, which suspends the computation and causes three "break" windows to appear on top of the lower edge of the typescript. The smallest window shows the dynamic state of the computation, which has been broken inside a subprogram called* FOLLOW/CURSOR. *The "*FOLLOW/CURSOR *Frame" window to the right shows the value of the local variables bound by* FOLLOW/CURSOR. *One of them has been selected (and so appears inverted) and in response, its value has been shown in more detail in the window at the lower left of the screen. The user has marked one of the component values as suspicious by drawing on it using the mouse. In addition, he has asked to examine the contents of the* BITMAP *component, which has opened up a bitmap edit window to the right. This shows an enlarged copy of the actual* NP *image that is being moved by the tree editor.*

Inside the largest break window, the user has asked some questions about FOLLOW/CURSOR, *and queried the value of* DRAWBETWEEN *(now 66). The* SHOW PATHS *command brought up the horizontal tree diagram on the left, which shows which subprograms call each other, starting at* FOLLOW/CURSOR. *Each node in the call tree produced by the* SHOW PATHS *command is an active element which will respond to the user's selecting it with the mouse. In the second image, the user has selected the* SHOWNODE *subprogram, which has caused its code to be retrieved from the file (*<LISP>DEMO>LATTICER*) on the remote file server (*PHYLUM*) where it was stored and displayed in the "Browser printout window" which has been opened at middle right. User programs and extended Lisp forms (like* **for** *and* **do***) are highlighted by system generated font changes. By selecting nodes in the* SHOW PATHS *window, the user could also have edited or obtained a summary description of any of the subprograms.*

Instead, the user has asked (in the break typescript window) to edit wherever anyone calls the DRAWBETWEEN *program (which draws lines between two specified points). This request causes the system to consult its (dynamically maintained) database of information about user programs, wherein it finds that the subprogram* SHOWLINK *calls* DRAWBETWEEN. *It therefore loads the code for* SHOWLINK *into an edit window which appears under the "Browser print out window". The system then automatically finds and underlines the first (and only) call on* DRAWBETWEEN. *On the previous line,* DRAWBETWEEN *is used as a variable (the one the user set and interrogated earlier). The system, however, knows that this is not a subprogram call, so it has been skipped. If the user makes any change to* SHOWLINK *in the editor, not only will the change take effect immediately, but* SHOWLINK *will be marked as needing to be updated in its file and the information about it in the program database will be updated. This, in turn, will cause the* SHOW PATHS *window to be repainted, as its display may no longer be valid.*


Top level typescript window
NIL
71←(DEFINEQ (F (A) (IF A LT 2 THEN 1 ELSE A*(F A-1]
(F)
72←(F 4]
24
73←FILES?]
LATTICER...to be dumped.
plus the functions: F
want to say where the above go ? No
NIL
74←(SETQ DRAWBETWEEN 0]
(DRAWBETWEEN reset)
0
75←(EDITTREE (PARSE My uncle's story
FOLLOW/CURSOR Frame
FOLLOW/CURSOR
ND {LNODE}#4,54110
DS {DISPLAYSTREAM}#5,137346
FOLLOW/CURSOR break: 1
(FOLLOW/CURSOR broken)
77:. SHOW PATHS FROM FOLLOW/CURSOR
NIL
78:. DOES FOLLOW/CURSOR CALL DRAWBETWEEN SOMEHOW
T
79:. DOES FOLLOW/CURSOR CALL DRAWBETWEEN
NIL
80:DRAWBETWEEN
66
81:
APPLY
EDITLATTICE
EDITTREEB0010
ADV←PROG
ADV←SETQ
PROG
EDITTREE
TOP
S
NP
AUX
VP
DET
N
PP
M
V
NP
PP
P
NP
PRO
P
NP
DET
N
N
PRO
My uncle's story about the war will bore you to tears
FOLLOW/CURSOR
NODE/POSITION
MOVE/NODE
MARKNODE
CHANGE/LAYOUT/POSITION
SHOWNODE
FROMLLINKS
NODEIDTONODE
SHOWLINK
TOPOS
FROMPOS
TOLLINKS
SHOW/NODE/LABEL
LABEL/IMAGE
BOTTOM/LEFT/CORNER
{LNODE}#4,54170 inspector
LNODEID (NP (DET &) (N &))
LNODEPOSITION (232 . 183)
NODELABELBITMAP {BITMAP}#5,43226
NODEFROMPOS (232 . 175)
NODETOPOS (232 . 191)
LNODEWIDTH 21
LNODEHEIGHT 16
TOLNODES ((DET &) (N &))
FROMLNODES ((PP & &))
LNODEFONT {FONTDESCRIPTOR}#1,115550
NODELABEL NP
BOXNODEFLG NIL
Bitmap Editor
NP
8-AUG-82
Interlisp-D


Top level typescript window
NIL
71←(DEFINEQ (F (A) (IF A LT 2 THEN 1 ELSE A*(F A-1]
(F)
72←(F 4]
24
73←FILES?]
LATTICER...to be dumped.
plus the functions: F
want to say where the above go ? No
NIL
74←(SETQ DRAWBETWEEN 0]
(DRAWBETWEEN reset)
0
75←(EDITTREE (PARSE My uncle's story
FOLLOW/CURSOR Frame
FOLLOW/CURSOR
ND {LNODE}#4,54110
DS {DISPLAYSTREAM}#5,137346
FOLLOW/CURSOR break: 1
78:. DOES FOLLOW/CURSOR CALL DRAWBETWEEN SOMEHOW
T
79:. DOES FOLLOW/CURSOR CALL DRAWBETWEE
NIL
80:DRAWBETWEEN
66
81:. EDIT WHERE ANY CALLS DRAWBETWEEN
SHOWLINK :
(DRAWBETWEEN (FROMPOS FRND) (TOPOS TOND
Browser print out window
{from {PHYLUM}<LISP>DEMO>LATTICER.;18}
(SHOWNODE
[LAMBDA (ND NODELST DS TOSONLY)
(* rrb "28-JAN-82 15:04")
(* displays a node and its links.
IF TOSONLY IS NON-NIL, DRAWS ONLY THE TO LINKS.)
(SHOW/NODE/LABEL ND DS)
(for TONODEID in (TOLLINKS ND) do (SHOWLINK ND (NODEIDTONODE TONODEID NODELST)
DS))
(OR TOSONLY (for FROMNDID in (FROMLLINKS ND) do (SHOWLINK (NODEIDTONODE FROMNDID
ND DS])
DEdit of function SHOWLINK
(LAMBDA (FRND TOND DS)
(* bas: "7-OCT-82 14:26")
(* draws in a link from
FRND TO TOND)
(add DRAWBETWEEN 1)
(DRAWBETWEEN (FROMPOS FRND)
(TOPOS TOND)
1 NIL DS])
EditOps
After
Before
Delete
Replace
Switch
( )
( ) out
Undo
Find
Swap
Reprint
DEdit
EditCom
Eval
Exit
APPLY
EDITLATTICE
EDITTREEB0010
ADV←PROG
ADV←SETQ
PROG
EDITTREE
TOP
S
NP
AUX
VP
DET
N
PP
M
V
NP
PP
P
NP
PRO
P
NP
DET
N
N
FOLLOW/CURSOR
NODE/POSITION
MOVE/NODE
MARKNODE
CHANGE/LAYOUT/POSITION
SHOWNODE
FROMLLINKS
NODEIDTONODE
SHOWLINK
TOLLINKS
SHOW/NODE/LABEL
LABEL/IMAGE
BOTTOM/LEFT/CORNER
{LNODE}#4,54170 inspector
LNODEID (NP (DET &) (N &))
LNODEPOSITION (232 . 183)
NODELABELBITMAP {BITMAP}#5,43226
NODEFROMPOS (232 . 175)
NODETOPOS (232 . 191)
LNODEWIDTH 21
LNODEHEIGHT 16
TOLNODES ((DET &) (N &))
FROMLNODES ((PP & &))
LNODEFONT {FONTDESCRIPTOR}#1,115550
NODELABEL NP
BOXNODEFLG NIL
Interlisp-D

transformation. Note that, usually, performance engineering takes place not as a single "post functionality optimization phase", but as a continuous activity throughout the development, as different parts of the system reach design stability and are observed to be performance critical. This is the method of "progressive constraint", the incremental addition of constraints as and when they are discovered and found important, and is a key methodology for exploratory development.

Both of these concerns can be most clearly seen in the various Lisp based systems. While, like all exploratory environments, they are often used to write code very quickly without any concern for efficiency, they are also used to write artificial intelligence programs whose applications to real problems are very large computations. Thus, the ability to make these programs efficient has long been of concern, because without it they would never be run on any interesting problems. More recently, the architectures of the new, personal Lisp machines like the 1108 have enabled fast techniques for many of the operations that are relatively slow in a traditional implementation. Systems like Interlisp-D, which is implemented entirely in Lisp, including all of the performance critical system code such as the operating system, display software, device handlers, *etc.*, show the level of efficiency which is now possible within an exploratory language.

PROSPECTS

The increasing importance of applications which are very poorly understood, both by their clients and by their would-be implementors, will make exploratory development a key technique for the 1980s. Radical changes in the cost of computing power have already made exploratory development systems cost effective vehicles for the *delivery* of application systems in many areas. As recently as five years ago, the tools and language features we have discussed required the computational power of a large mainframe (~$500K). Two years ago, equivalent facilities became available on a personal machine for ~$100K. A year later, ~$50K. Now, a full scale exploratory development system can be had for ~$25K. For many applications, the incremental cost has become so small over that required to support conventional technology that the benefits of exploratory development (and redevelopment!) are now decisive.

One consequence of this revolutionary change in the cost-effectiveness of exploratory systems is that our notion of "exploratory problem" is going to change. Exploratory programming was developed originally in contexts where change was *the* dominant factor. There is, however, clearly a *spectrum* of specification instability. Traditionally, the cost of exploratory programming systems, both in terms of the computing power required and the run-time inefficiencies incurred, confined their use to only the most volatile applications. Thus, the spectrum was arbitrarily dichotomized into "exploratory" (very few) and "standard" (the vast majority). Unfortunately, the reality is that unexpected change is far more common in "standard" applications than we have been willing to admit. Conventional programming techniques strive to preserve a stability that is only too often a fiction. Since exploratory programming systems provide tools that are better adapted to this uncertainty, many applications, such as office information systems, which are now being treated as "standard" but which in fact seem to require moderate levels of ongoing experimentation, may turn out to be more effectively developed in an exploratory environment.

We can also expect to see a slow infusion of exploratory development techniques into conventional practice. Many of the programming tools of an exploratory programming system (in particular, the information gathering and viewing tools) do not depend on the more exploratory attributes of either language or environment and could thus be adapted to support programming in conventional languages like FORTRAN and COBOL. Along with these tools will come the seeds of the exploratory perspective on language and system design, which will gradually be incorporated into existing programming languages and systems, loosening some of the bonds with which these systems so needlessly restrict the programmer.

To those accustomed to the precise, structured methods of conventional system development, exploratory development techniques may seem messy, inelegant and unsatisfying. But it's a question of congruence: Precision and inflexibility may be just as disfunctional in novel, uncertain situations as procrastination and vacillation are in familiar, well-defined ones. Those who admire the massive, rigid bone structures of dinosaurs should remember that jellyfish still enjoy their very secure ecological niche.

ACKNOWLEDGEMENTS

Many of these ideas were first developed, and later much polished, in discussions with John Seely Brown and other colleagues in Cognitive and Instructional Sciences at Xerox PARC.

The Interlisp programming environment

Warren Teitelman and Larry Masinter

Abstract

Interlisp is a programming environment based on the Lisp programming language. It has been used to develop and implement a wide variety of large application systems, primarily in the area of artificial intelligence. It is used by more than 300 researchers at 20 different sites in the US and abroad.

Interlisp provides an extensive set of user facilities, including syntax extension, uniform error handling, automatic error correction, an integrated structure-based editor, a sophisticated debugger, a compiler, and a filing system. This article describes the environment, the facilities available in it, and some of the reasons why Interlisp has developed as it has.

OVERVIEW

Interlisp is a programming environment based on the Lisp programming language. It is widely used within the artificial intelligence community, where it has been used to develop a variety of large application systems, including the Mycin system for infectious disease diagnosis [Shortliffe, 76], the Boyer-Moore theorem prover [Boyer and Moore, 79], and the BBN speech understanding system [Wolf and Woods, 80], *etc.*.

Interlisp supports this type of application development [Charniak *et al.*, 80] with an extensive set of user facilities, including syntax extension, uniform error handling, automatic error correction, an integrated structure-based editor, a sophisticated debugger, a compiler, and a filing system. Its most popular implementation is Interlisp-10, which runs under both the Tenex and Tops-20 operating systems for the DEC PDP-10 family. Interlisp-10 has approximately 300 users at 20 different sites (mostly universities) in the US and abroad and is an extremely well documented and maintained system.

From its inception, the focus of the Interlisp project has been not so much on the programming language as on the programming environment. An early paper on Interlisp [Teitelman, 69] states,

> In normal usage, the word "environment" refers to the aggregate of social and cultural conditions that influence the life of an individual. The programmer's environment influences, and to a large extent determines, what sort of problems he can (and will want to) tackle, how far he can go, and how fast. If the environment is cooperative and helpful (the anthropomorphism is deliberate), the programmer can be more ambitious and productive. If not, he will spend most of his time and energy fighting a system that at times seems bent on frustrating his best efforts.

The environmental considerations were greatly influenced by the perceived user community and the style of programming in that community: first, typical Lisp users were engaged in experimental rather than production programming; second, they were willing to expend computer resources to improve human productivity; third, we believed users would prefer sophisticated tools, even at the expense of simplicity.

EXPERIMENTAL PROGRAMMING AND STRUCTURED GROWTH

The original architects of the Interlisp system were interested in large artificial intelligence application programs. Examples of such programs are theorem provers, sophisticated game-playing programs, and speech and other pattern recognition systems. These programs are characterized by the fact that they often cannot be completely specified in advance because the problems - to say nothing of their solutions - are simply not well enough understood. Instead, a program must evolve as a series of experiments, in which the results of each step suggest the direction of the next. During its evolution, a program may undergo drastic revisions as the problem is better understood. One goal of Interlisp was to support this style of program development, which Erik Sandewall has termed *structural growth*:

> An initial program with a pure and simple structure is written, tested, and then allowed to grow by increasing the ambition of its modules. The process continues recursively as each module is rewritten. The principle applies not only to input/output routines but also to the flexibility of the data handled by the program, sophistication of deduction, the number and versatility of the services provided by the system, etc. The growth can occur both 'horizontally' through the addition of more facilities, and 'vertically' through a deepening of existing facilities and making them more powerful in some sense. [Sandewall, 78]

Sandewall's excellent survey article [Sandewall, 78] gives an overview of existing programming methodology in the Lisp environment, emphasizing methods for interactive program development. It includes a comprehensive description and analysis of current Lisp programming environments in general, and Interlisp and MacLisp in particular.

COMPUTER COSTS v. PEOPLE COSTS

The second major influence in Interlisp's development was a willingness to "let the machine do it." We were willing to expend computer resources to save people resources because computer costs were expected to continue to drop. This perspective sometimes led to tools which were ahead of their time with respect to the available computer resources.

The Advanced Research Projects Administration of the Department of Defense sponsored much of the early work on Interlisp. Their willingness to make Interlisp available at a number of sites on the Arpanet justified and motivated the extra effort it took to turn a research project into a real system. These Arpanet sites also provided an active and creative user community from which we obtained many valuable suggestions and much-needed feedback.

INTERLISP WAS FOR EXPERTS

The incremental, evolutionary way in which Interlisp developed was not especially conducive to simple interfaces. It was inappropriate to spend a lot of time and effort trying to design the right interface to a new, experimental capability whose utility had not yet been proven. Would the users like automatic error correction? Was the programmer's assistant really a good idea? The inherent complexity of the interactions among some of the more sophisticated tools, such as Masterscope, DWIM, and the programmer's assistant, made it very difficult to provide simple interfaces. In many cases, unification and simplification came only after considerable experience.

Further complexity stemmed from the commitment to accommodate a wide variety of programming styles and to enable the tools to be tailored for many applications. Given a choice of sophistication and generality of tools or simplicity of design, we chose the former, under the assumption that the system was primarily for expert programmers. As a result, mastery of all of the facilities of Interlisp has become quite difficult and initial learning time fairly long. We accept this as part of the price for the system's power and productivity.

BACKGROUND

Programming environments have been built for a number of languages, on top of a number of operating systems, and for a variety of user communities. Each of these factors can influence the path taken in the development of a programming environment. In the case of Interlisp, the Lisp language itself and the sociological factors in effect during its early development were both important.

The Lisp Language

The Lisp language is conducive to the development of sophisticated programming tools because it is easy to write programs that manipulate other programs. The core syntax for the Lisp language is simple, and Lisp programs are naturally represented in simple Lisp data structures in a way that reflects the structure of the program. Since Lisp requires no declarations, programs can be built up incrementally; this is more difficult in declarative languages. This means that Lisp supports the structured growth style of program building.

Early Sociology of Interlisp

One unusual historical aspect of the development of Interlisp is that from the very beginning those interested in programming environments were in a position to strongly influence the development of the language system. We were not constrained to live within the language and operating system we were given, as is usually the case. Most of the additions or extensions to the underlying Lisp language performed under the Interlisp project were in response to perceived environmental needs. For example, Interlisp permits accessing the control stack at an unusually detailed level. Capabilities such as this were added to Interlisp to enable development of sophisticated and intelligent debugging facilities. Similarly, uniform error handling was added to the Lisp base in order to permit experimentation with automatic error correction.

SOME REPRESENTATIVE FACILITIES

File Package

Interactive program development consists alternately of testing program parts and editing them to correct errors discovered during the tests and/or to extend the program. Interlisp, unlike many other interactive programming systems, supports both testing and editing operations. The user talks exclusively to the Interlisp system during the interactive session. During this process, the primary copy of the program (the copy that is changed during editing operations) resides in the programming system as a data structure; editing is performed by modifying this data structure. For this reason, Interlisp is called a *residential system* [Sandewall, 78].

In a residential system, one must be able to take procedures represented by data structures and print them on text files in an input-compatible format for use as backup, to transport programs from one environment to another, and to provide hard-copy listings. The Interlisp file package is a set of functions, conventions, and interfaces with other system packages which facilitate this by automating the bookkeeping necessary to maintain a large system consisting of many source files and their compiled counterparts. The file package removes from the user the burden of keeping track of where things are and what things have changed. For example, the file package keeps track of which file contains each datum, *e.g.*, a function definition or record declaration. In many cases, it automatically retrieves the necessary datum, if it is not already in the user's working environment. The file package also keeps track of which files have been in some way modified and need to be dumped, which files have been dumped but still need to be recompiled, *etc.*. [Teitelman *et al.*, 78] Once the user agrees to operate in the residential mode, it becomes possible to design and implement such powerful tools as DWIM and Masterscope to assist in program development. The file package makes this mode attractive to the user.

The history of the file package is instructive, as it is a paradigm for the development of user facilities that has frequently been followed in Interlisp. The file package was *not* designed in a coherent, integrated way; nobody sat down and said, "We need a file package." Instead, it evolved gradually. Originally, there was only a very limited facility for symbolically saving the state at the end of a session in a form that could be loaded into a Lisp system to restore that state: the PRETTYDEF function. This took as its arguments a list of function names, a list of variable names, and a file name. PRETTYDEF wrote ("prettyprinted") the definitions of the named functions and the values of the named variables onto the indicated file. PRETTYDEF was soon extended to take a set of commands, which could indicate not only the functions and variables to be saved, but properties on property lists, values in arrays, definitions of new editor commands, and record declarations, among others. Finally, PRETTYDEF was extended to allow the user to augment this simple command language by defining his own filing commands (usually in terms of existing ones).

Concurrently, as the contents of source files became more complicated, the ability to interrogate files as to their contents (*e.g.*, which file contained a particular datum and what functions were contained in a particular file) became more important. This required that the system be able to enumerate all of the user's source files, which was accomplished by adding the file to a global list so it would be noticed when it was first loaded or dumped.

The significant breakthrough occurred with the emergence (probably through some user saying, "Wouldn't it be nice if...") of the idea of having the system notice when a datum was changed, *e.g.*, defined for the first time, edited, redefined, or reset, and associate this fact with the file containing the datum. This enhancement was relatively straightforward since the ability to decompose and interpret the commands that described the contents of a particular file was already available. It was implemented by a function that took the name of a datum and its type (function, variable, record definition, *etc.*) and marked the datum as changed and therefore in need of dumping. This function, MARKASCHANGED, determined which file(s) contained the changed datum and associated with that file the name of the object that had changed. Calls to MARKASCHANGED were then inserted wherever the Interlisp system changed objects - the editor, the DEFINE function, the facility for (re)declaring records, and DWIM (which can modify a function by making a spelling correction or some other transformation).

With this change, the file package assumed a degree of autonomy, often operating automatically and behind the scenes. Furthermore, since it was no longer a function that the user called, but included "tendrils" into many parts of the system, we began to think of it as a package. In this light, a number of extensions became apparent. For example, the CLEANUP function provided the capability to enumerate all of the user's files and write out those containing objects that had changed. Next, an automatic warning was added to CLEANUP, in case an object not associated with any file changed or was newly defined. Then a filing capability was added, which enabled the user to add a datum to a file by automatically modifying the commands for that file. Finally, CLEANUP was extended to ask about unfiled objects and to allow the user to specify where they should go.

By this point, the Interlisp user did not have to worry about maintaining his source files, save for occasionally calling CLEANUP. The file package had become smart with respect to its built-in commands, but if a user defined a new type of command, the file package would not necessarily be able to support operations such as adding an object of that type to a file, deleting or renaming an object, or obtaining the "definition" of an object of a particular type from a file. The user was placed in the position of having to choose between using an automatic facility that did just what he wanted - provided he stuck to a predefined class of file objects - or extending this facility to print out his own types of file objects, which meant returning to manual bookkeeping of symbolic files.

Thus, the next extension to the file package identified and exposed its primitive operations and allowed the user to define or change these operations. This resulted in a more complicated interface to the file package than that used by most, but it enabled builders of systems within Interlisp to enjoy the same privileges of defining file package operations that the original implementors enjoyed. In fact, we were able to express the semantics of all of the built-in file package commands and types in terms of the above interface. We thus eliminated all distinction between built-in operations and those defined by the user (a good test of the completeness of this lower-level interface) and permitted the user to redefine the way these operations are performed.

The file package supports the abstraction that the user is truly manipulating his program as data and that the file is merely one particular external representation of a collection of program pieces. During the session, the user manipulates the pieces with a variety of tools, occasionally saving what he has done by calling CLEANUP. The user can also operate in a mode where programs are treated as residing in a data base, *i.e.*, the external file system, with a variety of sophisticated retrieval tools at his disposal.

Note the evolution of the file package. It started as an isolated facility that was explicitly invoked by the user to perform a particular and limited action. More and more capabilities were added, increasing the range of applicability of the tool. At the same time, the tool was integrated into the system to produce a semi-autonomous configuration in which the tool is invoked automatically in a number of contexts. Finally, the utility of the tool became so great that a form of user extensibility, to adapt the tool to accommodate unforeseen situations, became imperative.

The file package also illustrates one of the principal design criteria of the Interlisp system, the accommodation of a wide range of styles and applications. The user is not forced to

choose among using a facility that is powerful and attractive but forces adherence to its prescribed conventions, abandoning the tool, or even creating a personal, renegade version whenever he needs a capability the tool does not provide. In other words, if a particular tool handles 95 percent of the user's applications correctly, he should be able to extend the tool in a prescribed and "blessed" manner to accommodate the remaining five percent without undue effort.

Masterscope

As the size of systems built within Interlisp grew larger and larger, it became increasingly difficult for a user to predict the effect of a proposed change. It was also growing difficult to effect a pervasive change, for example, to change the calling convention of a low-level procedure and be sure that all of the relevant places in programs would be found and modified. Masterscope is an interactive subsystem for analyzing and cross-referencing user programs that addresses this problem. It contains facilities for analyzing user programs to determine which functions are called, how and where variables are bound, set, or referenced, which functions use particular record declarations, etc.

Masterscope maintains a data base of the results of the analyses it performs. The user can interrogate the data base explicitly (*e.g.*, WHO USES FOO FREELY), or have Masterscope call the editor on all functions that contain expressions that satisfy certain relations (EDIT WHERE ANY FUNCTION USES THE RECORD DICTENTRY).

Masterscope, like the file package, has its roots in an extremely simple program. Called **PrintStructure**, this program analyzed function definitions and printed out the tree structure of their calls. It was first extended to include the names of the arguments for each function it analyzed, and then to include more information about variable usage within each function. However, as **PrintStructure** presented more and more information about larger and more complicated program configurations, it became increasingly difficult for the user to extract particular information from this massive output. It became clear that the user wanted access to specific information rather than a complete listing. This led to the idea of separating the analysis of the program from the interrogation of the data base.

The next stage was integration with the other parts of the system. As in the case of the file package, the utility of Masterscope increased greatly when the burden of remembering what had changed, and therefore needed re-analysis, was lifted from the user and carried out automatically behind the scenes. The next phase of the evolution of Masterscope was to permit the user to extend Masterscope's built-in information on analysis of special Lisp forms, such as PROG, SETQ, and LAMBDA expressions. This was accomplished through the use of Masterscope "templates," which are essentially patterns for evaluation of functions. Finally, all built-in information was removed from Masterscope and replaced by templates, both to test the completeness of the interface and to expose this information to users so they could change it.

DWIM

> One of the most impressive features in the Interlisp system is the DWIM (Do What I Mean) facility, which is invoked when the basic system detects an error and which attempts to guess what the user might have intended. [Sandewall, 78]

The most visible part of DWIM [Teitelman, 72a] is the spelling corrector, which is invoked from many places in the system, including the file package, Lisp editor, and the Lisp interpreter itself. When an unrecognized file package command, edit command, Lisp function, *etc.*, is encountered, the spelling corrector is invoked. The spelling corrector attempts to find the closest match within a list of relevant items. If an edit command is misspelled, for example, the list of valid edit commands is searched; if the name of a function is misspelled, the corrector scans a list of the functions the user has recently been working with. If the spelling correction is successful, the cause of the error is also repaired, so subsequent corrections will not be necessary. For example, when DWIM corrects a user's misspelled function name in one of his programs, it actually modifies the user's program to contain the correct spelling (and notifies the file package of the change).

Although most users think of DWIM as a single identifiable package, it embodies a pervasive philosophy of user interface design: at the user interface level, system facilities should make reasonable interpretations when given unrecognized input. Spelling correction is only one example of such an interpretation. Depending on how far off the input is, a facility might make the transformation silently and automatically, without seeking user approval. For example, a function expecting a list of items will normally interpret an argument that is a single atom as a list made up of that single atom. In this case, the package in question probably would not even indicate to the user that it had made this correction, and in fact the user might view the package as expecting either a list or an atom. Similarly, the style of interface used throughout Interlisp allows the user to omit various parameters and have these default to reasonable values, such as "the last thing this package operated upon."

DWIM is an embodiment of the idea that the user is interacting with an agent who attempts to interpret the user's request for contextual information. Since we want the user to feel that he is conversing with the system, he should not be stopped and forced to correct himself or give additional information in situations where the correction or information is obvious.

The Iterative Expression

The various forms of the Interlisp iterative expression permit the user to specify complicated loops in a straightforward and visible manner. In one sense, the iterative expression represents a language extension, but by its design, implementation, and in particular its extensibility, it more naturally falls into the same category as other Interlisp tools.

An iterative expression in Interlisp consists of a sequence of operators, indicated by keywords, followed by one or more operands; many different operators can be combined in the same iterative statement. For example, (for X in L sum X) iterates the variable X over the elements of the list L, returning the sum of each value seen. The iterative expression could be further embellished by including "when (GREATERP X 30)" to only sum elements greater than 30, or "while (LESSP $VAL 50)" to terminate the iteration when the sum exceeds 50. Other operators can be used to specify different ranges. For example, iteration can take place over a range of numbers instead of over the elements of a list, *e.g.*, (for I from 1 to 10 ...). Operators can also specify the value returned by the iterative expression. For example, (for X in L collect (ADD1 X)) would return a new list, consisting of the elements in L, each incremented by 1.

The iterative expression currently understands approximately two dozen operators. Furthermore, new iterative operators can be defined simply. One group, experimenting with a relational data base system, provided access to that data base merely by defining a new iterative operator called "matching." This matching operator can be used in conjunction with all of the other iterative constructs, as in "(for Records matching (PAYMENT (> 30) *) sum Record:3)," which would find all payment records in the data base and sum their third component. Such language extensions are quite difficult in most programming languages.

Programmer's Assistant

The central idea of the programmer's assistant is that the user is not talking to a passive executive that merely responds to each input and waits for the next, but is addressing an active intermediary [Teitelman, 72b]. The programmer's assistant records, in a data structure called the history list, the user's input, a description of the side effects of the operation, [Teitelman *et al.*, 78] and the result of the operation.

The programmer's assistant also responds to commands that manipulate the history list. For example, the REDO command allows the user to repeat a particular operation or sequence of operations; the FIX command allows the user to invoke the Interlisp editor on the specified events and then to re-execute the modified operations; the USE command performs a substitution before re-executing a specified event (e.g., USE PRINT FOR READ); the UNDO command cancels the effect of the specified operations. In addition to the obvious use of recovering information lost through typing errors, UNDO is often used to selectively flip back and forth between two states. For example, the user might make some changes to his program and/or data structures, run an experiment, undo the changes, rerun the experiment, undo the undo, and so on.

The various replay commands, such as REDO and FIX, permit the user to construct complex console operations out of simpler ones, in much the same fashion as programs are constructed. That is, simple operations can be first checked and then combined into large ones. The system always remembers what the programmer has typed, so that keyboard input can be reused in response to an afterthought.

The programmer's assistant has been implemented for use in contexts besides the handling of inputs to the Lisp "listen" loop. For example, the Interlisp editor also uses the programmer's assistant for storing operations on the history list and thereby provides all the history commands for use in an editing session. Similarly, user programs can take advantage of the history facility. A system for natural language queries of a data base of lunar rock samples provides one example of how this facility can be used. After a complicated query regarding the percentage of cobalt in a sample, a user could say USE MANAGANESE FOR COBALT to repeat the query with a different parameter.

WHAT MAKES INTERLISP UNIQUE?

The Interlisp programming environment has been characterized as friendly, cooperative, and forgiving. While these qualities are desirable, they are not unique to Interlisp. The two attributes that set it apart are the degree to which the system is integrated and the degree to which facilities in the environment can be tailored, modified, or extended.

Integration

Interlisp is not merely a collection of independent programming tools, but an integrated system. By integration, we mean that there need not be any explicit context switch when switching between tasks or programming tools, in switching, for example, from debugging to editing to interrogating Masterscope about the program. Thus, having called the editor from inside the debugger, the user can examine the current run-time state from within the editor or ask Masterscope a question without losing the context of the editing session. Also, the various facilities themselves can use each other in important ways, since they all coexist in the same address space. For example, the editor can directly invoke DWIM, or Masterscope commands can be used to drive the editor. The integration of facilities increases their power.

Integrated programming tools such as these are not feasible without a large virtual address space. Where the size of the programming environment is constrained, it is unreasonable to have a large variety of resident tools that can all interact with the user's run-time environment and with each other. Interlisp-10's large virtual address space of 256K 36-bit words (large, at least for the early 1970's) made it possible to add new features to the programming environment without trying to squeeze them into a small amount of space or worrying about leaving enough space for the user.

Extensibility

Most programming environments, even when they provide a variety of tools, support only a narrow range of programming styles. In the development of Interlisp, we have tried to accommodate a variety of programming styles.

The most straightforward way of allowing users to modify or tailor system tools to their own applications is simply to make sources available and allow the users to edit and modify tools as they wish. A benefit of this approach is that it absolves the system designers of responsibility for unforeseen bugs or incompatibilities. ("The manufacturer's warranty is void if this panel is removed.") Of course, this kind of extensibility isn't really defensible, as it discourages all but the most intrepid of users. If a creative user does manage to extend a system capability, he must then worry about tracking improvements and bug fixes in this tool and be constantly aware of changes to the system, which could introduce incompatibilities with respect to his modifications.

Extensions and modifications were provided in a variety of ways. Capabilities that have associated command languages lend themselves quite naturally to extensibility, because new commands can be defined in terms of existing ones. Almost all Interlisp packages (e.g., the file package, the editor, the debugger and programmer's assistant) support such extensions via substitution macros, which associate a template (composed of existing commands) with the new command. The arguments to the new command are then substituted for those of the template's as appropriate. In addition, most facilities support computed macros. A computed macro is basically a Lisp expression, evaluated to produce a new list of operators, commands, or expressions. For example, a computed edit macro produces a list of edit commands, but a computed file package macro produces a list of file package commands.

However, many extensions are not expressible in terms of macros because they are triggered not by the appearance of a particular token, but by the existence of a more general condition. Interlisp provides for such extensions by allowing the user to specify a function to be called upon any object/expression/command that the particular facility does not recognize. This function is responsible for selecting from among the various conditions that might pertain and deciding whether or not it recognizes a particular case. If it does, it takes the appropriate action. Typical applications of such functions are implementation of infix edit commands and specification of the compilation of a class of expressions, such as the iterative expression.

For example, the DWIM facility, which corrects spelling errors encountered while running, is implemented via an extension to the Lisp interpreter of this form, called FAULTEVAL. Whenever the Interlisp interpreter encounters an expression for which it is going to generate an error, such as an undefined function or variable, the interpreter instead calls FAULTEVAL. Originally, FAULTEVAL merely printed an error message. DWIM was implemented by redefining FAULTEVAL to try to correct the spelling of the undefined function or variable, according to names defined in the context in which the error occurred.

One might suppose that a facility as basic as correction of program errors would have been implemented by modifying the Lisp interpreter - especially since a fair amount of knowledge about the interpreter's state was required in order to be able to continue a computation after an error correction. The fact that this is not the case illustrates a basic tenet of the Interlisp design philosophy, which holds that the implementation of enabling capabilities is a top priority. When DWIM was first being implemented, the interpreter did not call FAULTEVAL, and there was no way to trap all DWIM errors. Instead of trying to implement DWIM directly, we tried to find the enabling capability that would make it possible for a *user* to implement DWIM. This capability was provided by having the interpreter call FAULTEVAL, which was then used to implement DWIM.

The enabling capability was then available for other applications, as well. It has allowed users to experiment with building their own tools and extending system capabilities in ways we did not foresee. For example, one application program redefined FAULTEVAL to send error messages not to the user but instead to the implementor of the application, via computer mail.

Finally, because we realized that some users just might not like a particular facility, we made it easy for them to "turn off" any automatic facility in the system. This made the use of the programming tool a deliberate choice of the user and provided a powerful force for quality control: if the feature didn't help as much as it got in the way, people would turn it off.

The support of a wide variety of programming styles and settings of parameters has some drawbacks. Interlisp has an overabundance of user-setable parameters, to the point where new users are sometimes overwhelmed by the number of choices. In addition, it is necessary to ensure that the system will work correctly for every possible setting of the various system parameters. For example, the Masterscope facility normally relies on DWIM to perform some of its transformations, so we had to take care that Masterscope would continue to work, even if the user disabled DWIM.

A BRIEF HISTORY OF INTERLISP

Interlisp began with an implementation of the Lisp programming language for the PDP-1 at Bolt, Beranek and Newman in 1966, followed in 1967 by 940 Lisp, an upward compatible implementation for the SDS-940 computer. 940 Lisp was the first Lisp system to demonstrate the feasibility of using software paging techniques and a large virtual memory in conjunction with a list-processing system [Bobrow and Murphy, 67]. 940 Lisp was patterned after the Lisp 1.5 implementation for CTSS at MIT, with several new facilities added to take advantage of its timeshared, on-line environment.

The SDS-940 computer was soon outgrown, and in 1970 BBN-Lisp, an upward compatible version of the system for the PDP-10, was implemented for the Tenex operating system. With the hardware paging and 256K of virtual memory provided by Tenex, it was practical to provide more extensive and sophisticated user support facilities, and a library of such facilities began to evolve. In 1972, the name of the system was changed to Interlisp, and its development became a joint effort of the Xerox Palo Alto Research Center and Bolt, Beranek and Newman. The next few years saw a period of rapid growth and development at the language and system levels and at user support facilities, notably in the record package, the file package, and Masterscope. This growth was paralleled by the increase in the size and diversity of the Interlisp user community.

In 1974, Interlisp was implemented for the Xerox Alto, an experimental microprogrammed minicomputer [Thacker *et al.*, 79]. AltoLisp introduced the idea of providing a microcoded target language for Lisp compilations, which modelled the basic operations of Lisp more closely than could a general-purpose instruction set [Deutsch, 73a]. AltoLisp also served as a model and departure point for Interlisp-D [Burton *et al.*, 80] the implementation of Interlisp for the Dolphin and Dorado Xerox personal computers [Lampson and Pier, 80], the successors to the Alto. Interlisp-D now supports a large user community within Xerox Palo Alto Research Center.

Evolution of Interlisp

The origins of Interlisp at Bolt, Beranek and Newman were fortuitous. There was neither an existing Lisp implementation for the available hardware nor a user community, so it was necessary to start from scratch. Along with the necessity of starting from scratch came the freedom to develop the environment. We were free to experiment with various ideas and facilities, discard those that did not work out, and learn from mistakes in the process. We approached the problem of building the programming environment with the same paradigm with which we approached the programs being developed in that environment - as an ongoing research problem, not something that had to be right the first time or even finished at all. New capabilities were often introduced without a thorough design or a complete understanding of the underlying abstractions. Furthermore, "hooks" into the system were provided at many different levels in order to encourage users to augment system packages or experiment with their own. Many of the now-permanent facilities of the Interlisp system evolved from tools designed by individual users to augment their own working environments.

The result was a somewhat chaotic growth pattern and a style sometimes characterized as baroque. Interlisp was not designed, it evolved - but this was the right approach. As

Sandewall [Sandewall, 78] points out, "The task of designing interactive programming systems is hard because there is no way to avoid complexity in such systems.... The only applicable research method is to accumulate experience by implementing a system, synthesize the experience, think for a while, and start over." Had we been required to convince a disinterested third party of the need for certain enabling facilities in the language or operating system in order to perform experiments - whose exact shape and ultimate payoff were unknown - many of the more successful innovations would not exist. The value of a number of these innovations, including history, UNDO, and spelling correction, is now well recognized and accepted, and many new programming environments are being built with these facilities in mind.

The ability of individual users to augment system tools at a variety of levels, as well as quick responses to suggestions for extensions that users could not perform themselves, contributed greatly to the enthusiasm and energy of the Interlisp community. These factors played a large part in the growth and success of the system over the last decade.

Of course, as Interlisp matured and the user community grew, we were occasionally restricted in some areas of experimentation by a concern for backwards compatibility and the fact that the system was being used to get "real work" done. But enough flexibility had been built in to permit experimentation without performing major low-level changes. Furthermore, Interlisp attracted users who appreciated its flexibility and enjoyed experimentation with avant-garde facilities. Thus, when planned evolution led to some incompatibilities and consequent retrofitting, our user community was understanding and supportive.

FUTURE DIRECTIONS

Interlisp and personal computing

Interlisp evolved in a timeshared, hard-copy terminal world, and vestiges of this heritage have carried over into implementation for personal computers. In the future, we expect to see increasing exploitation of the personal nature of the computing environment. For example, there is a significant difference between performing an Interlisp-10 operation on a lightly loaded timeshared system and one that is heavily loaded. If the former takes 50 milliseconds, the latter might take as long as five seconds of real time, especially if the computation involves a large working set, as is often the case with the more sophisticated facilities of Interlisp. This makes the probability high that portions of the working set will be swapped out before the computation completes, and therefore must be swapped back in again, adding to the delay.

This real-time difference is especially relevant when dealing with interactive tools. A system can afford to spend 50 milliseconds trying to find out what a user means, because the extra 50 milliseconds is insignificant compared to the overhead of interacting with the user. But a system that spends five seconds to perform a spelling correction is often not acceptable, because in most situations the user would prefer to retype the correct input rather than wait. In such a case, the tool not only fails to add to the interactive quality of the system for this particular user, but since the user is competing with others for the same resource, namely machine cycles, its very attempt to be helpful causes response time - and hence the

interactive quality of the system - to degrade for other users. To quote Sandewall, "When this facility [DWIM] is presented to new users, it is not uncommon for them to use it for a trivial typing error that could easily be corrected using the character-delete key. However, the user relies on DWIM for the correction, which at periods of peak computer load may take considerable time.... As computer systems become more and more heavily loaded, more of the advanced features in interactive programming systems are canceled." [Sandewall, 78].

The entire situation changes in the personal computing environment. It is no longer necessary to justify the use of a particular tool by a single user in terms of the overall productivity of the community, since there is no longer any competition for cycles. It even becomes reasonable to devise tools that operate continually in a background mode while the user is thinking, such as an incremental garbage collector or programs that update a Masterscope data base. Personal computing is thus a qualitative change in the programming environment, because the machine is working continually for a single user.

Integration of the display

A significant addition to Interlisp in the new generation of personal computers, such as Interlisp-D, is the availability and integration of very high-resolution and high-bandwidth displays. Because of the high-output bandwidth of the display and the increased input bandwidth arising from the use of pointing devices, a number of trade-offs change significantly. The capabilities affect, for example, something as elementary as how much information to present to the user when an error occurs; the utility of on-line documentation assistance also increases. Complicated sequences of commands for specifying location, down to a particular frame on the stack, a particular expression in a program, *etc.*, are obviated by the ability to display the data structure and have the user point at the appropriate place. Similarly, the choice between short, easily typed, but esoteric command or function names as opposed to those that are longer, more self-explanatory, but more difficult to type becomes academic when operations can be invoked via menus.

These are examples of how a high-resolution display can facilitate essentially the same operations found in the hard-copy domain. Perhaps more interesting are the modes of operation enabled by the display that are unlike those of the hard-copy world. DLisp was an early experimental system that explored some of these techniques in Interlisp [Teitelman, 77]. In DLisp, the user sees his programming environment through a collection of display windows, each of which corresponds to a different task or context. The user can manipulate the windows, or the contents of a particular window, by a combination of keyboard inputs and pointing operations. The technique of using different windows for different tasks makes it easy for the user to manage several simultaneous tasks and contexts, *e.g.*, defining programs, testing programs, editing, asking the system for assistance, and sending and receiving messages. It also facilitates switching back and forth between these tasks.

Finally, we have not *(at the time of writing, November 1980, Ed.)* really begun to explore the use of graphics - textures, line drawings, scanned images, even color - as a tool for program development. For example, the system might present storage in a continually adjusting bargraph, or display a complicated data structure as a network of nodes and directed arcs, perhaps even allowing the user to edit this representation directly. This is a rich area for development in the future.

REFERENCES

Bobrow, D. G. & Murphy, D. L.
The structure of a Lisp system using two level storage. *Communications of the ACM*, 10:3 (March 1967), 155-159.

Boyer, R. S. & Moore, J. S.
A Computational Logic. Academic Press, New York, 79.

Burton, R. R.
Interlisp-D display facilities. In *Papers on Interlisp-D*, Xerox PARC Technical Report SSL-80-4, 1980, 33-46.

Burton, R. R. *et al.*
Interlisp-D: overview and status. In *Papers on Interlisp-D*, Xerox PARC Technical Report SSL-80-4, 1980, 1-10.

Charniak, E. *et al.*
Artificial Intelligence Programming. Lawrence Erlbaum Associates, Hillsdale, N.J., 80.

Deutsch, L. P.
A Lisp machine with very compact programs. *Proceedings of the third international joint conference on artificial intelligence,* Stanford 1973.

Lampson, B. W. and Pier, K. A.
A processor for a high-performance personal computer. *Seventh International Symposium on Computer Architecture*, La Baule, France, May 1980.

Sandewall, E.
Programming in the interactive environment: the Lisp experience. *Communications of the ACM*, 10:1 (March 1978), 35-71.

Shortliffe, E. H.
Computer-Based Medical Consultations. American Elsevier, New York, 76.

Teitelman, W.
Toward a programming laboratory. *International Joint Conference on Artificial Intelligence*, Washington, May 1969, 1-8.

Do What I Mean. *Computers and Automation*, April 1972.

Automated programming - the programmer's assistant. *Proceedings of the Fall Joint Computer Conference*, AFIPS Proceedings (1972), 917-922.

A display-oriented programmer's assistant. Xerox PARC Technical Report CSL-77-3, 77. See also *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, Massachusetts, August 1977, 905-915.

Teitelman, W. *et al.*
Interlisp Reference Manual. Xerox PARC, 1978.

Thacker, C. *et al.*
Alto: a personal computer. Xerox PARC Technical Report CSL-79-11, August 1979.

Wolf, J. & Woods, W.A.
The HWIM speech understanding system. In *Trends in Speech Recognition*, Prentice-Hall, Englewood Cliffs, New Jersey, 80.

Interlisp-D: Overview and Status

Richard R. Burton, Larry M. Masinter, Alan Bell, Daniel G. Bobrow,
Willie Sue Haugeland, Ronald M. Kaplan and B. A. Sheil

Abstract

Interlisp-D is an implementation of the Interlisp programming system on the Dolphin and Dorado, two large personal computers. It evolved from AltoLisp, an implementation on a less powerful machine. This paper describes the current status of Interlisp-D and discusses some of the issues that arose during its implementation. The techniques that helped us improve the performance included transferring much of the kernel software into Lisp, intensive use of performance measurement tools to determine the areas of worst performance, and use of the Interlisp programming environment to allow rapid and widespread improvements to the system code. The paper lists some areas in which performance was critical and offers some observations on how our experience might be useful to other implementations of Interlisp.

BACKGROUND

Interlisp is a dialect of Lisp whose most striking feature is a very extensive set of user facilities including syntax extension, error correction, and type declarations [Teitelman *et al.*, 78]. It has been in wide use on a variety of time shared machines over the past ten years.

AltoLisp

In 1974, an implementation of Interlisp for the Alto, a small personal computer, was begun at Xerox PARC by Peter Deutsch and Willie Sue Haugeland [Deutsch, 1973]. This AltoLisp implementation introduced the idea of providing a microcoded target language for Lisp compilations which modelled the basic operations of Lisp more closely than a general purpose instruction set. A similar instruction set was also implemented for Maxc, a microprogrammed machine running the TENEX operating system [Fiala, 1978].

The design of AltoLisp is presented in [Deutsch, 1978]. Its characteristics include a very large address space (24 bits); deep binding; CDR encoding [Bobrow & Clark, 1979]; transaction garbage collection [Deutsch & Bobrow, 1976]; and an extensive kernel implemented in a mix of microcode and Bcpl. Although AltoLisp was completed and several large Interlisp programs were run on it, its performance was never satisfactory, due principally to the limited amount of main memory and the lack of support in the processor architecture for either virtual memory management or byte code decoding. Interlisp-D is the result of transferring AltoLisp to an environment with neither of these limitations.

Interlisp-D

The Dorado [Lampson & Pier, 1980] is a large, fast, microcodable personal machine with 16-bit data paths. It has a large main memory (~1 megabyte) and hardware support for both instruction decoding and virtual memory management. The Dolphin is a similar, but smaller and less powerful, machine.

A revised version of a paper originally presented at the 1980 Lisp Conference, Stanford, Ca.

Both machines have microcode to emulate the Alto, so the initial transfer of the running AltoLisp system to them was straightforward. Although the microcode to interpret the Lisp instruction set needed to be rewritten, the Bcpl runtime support system was transported with only minor changes. However, initial performance was far worse than would be expected from a simple consideration of machine features. We expected Dorado Interlisp-D to dominate Interlisp-10 running on a single user DEC KA-10, but in fact, some computations took 10 to 100 times longer. Our primary goal, then, became to improve the performance of the existing system. First, careful measurements were taken of the system doing a variety of tasks. Functions which took inordinate amounts of time were examined in detail. Additional microcode was written, and major portions of the Lisp code were redone.

The most surprising thing to us was that we obtained considerable performance improvements by moving large parts of the system from Bcpl into Lisp. This allowed us to use a number of programming tools in the Interlisp system, and allowed us to put more structure into the layers of the system's kernel. Interlisp-D is now supporting a large user community. While speed ratios vary widely across different classes of computation, it appears that Dorado Interlisp-D runs more than five times faster than Interlisp-10 on a single-user DEC KA-10. *[In August 1980, Ed.]*

THE "LISPIFICATION" OF INTERLISP-D

Much of the Interlisp system is written in Lisp itself, resting on a kernel not defined in Lisp. The Interlisp virtual machine specification [Moore, 1976] attempted to identify a set of kernel facilities which would support the full Interlisp system. This was done by carefully documenting those parts of the PDP-10 Interlisp system that were written in assembly language or imported from the operating system. This specification is quite large. AltoLisp reduced this kernel by implementing some of the VM facilities in Lisp; Interlisp-D accelerated this development. In addition to improving the transportability of the implementation, the move also improved performance, gave the implementors access to more a more powerful implementation language and programming tools, and limited the breadth of expertise required of system implementors.

Efficiency

Programs written in a higher level language are often less efficient than equivalent assembly language programs, because they cannot exploit known invariances and optimizations which would violate the strict semantics of the target language. Moving code from Lisp into the kernel has been a traditional way of improving the performance of Lisp systems. Substantial sections of the PDP-10 implementation of Interlisp, for example, are in machine code for this reason. When a large proportion of AltoLisp was moved from Bcpl into Lisp in order to improve memory utilization and aid modification. the speed of the system decreased by nearly a factor of three [Deutsch, 1978]. Thus, to improve Interlisp-D performance, we first looked for Lisp-coded sections of the system that could be incorporated into the Bcpl kernel. However, we soon discovered that the poor performance was due more to the design of the algorithms in the kernel than to the language in which they were implemented. Since we did not wish to carry out a large-scale redesign in the limited Bcpl programming environment, we decided to go in the other direction: we would move code *out* of the extended Bcpl kernel and into Lisp so that we would be better able to change the algorithms. Specific targets for

replacement were large sections of the Bcpl kernel with known performance problems whose functionality could easily be expressed in Lisp; one of the major areas was the I/O system.

Language power and tools

A primary reason for implementing the bulk of a programming system in itself is that one obtains the advantage of programming in a (presumably) more expressive and powerful language. In addition, we felt that the major modifications and tuning that would be necessary to provide adequate performance would be far more tractable in Interlisp. In Interlisp we had both a first rate programming environment and instrumentation tools, and we had no other system implementation language which had either. Our subsequent experience has sustained this view.

Linguistic uniformity

An important sociological benefit of having a programming system described in the language it implements is that the system's implementors and users share the same culture. Users can inspect the system code, comment on it, adapt it for their own purposes, and sometimes even change it. This involves the users of the system in its design and maintenance in a way that would not be possible if system construction took place in a different language culture. Specifically, the availability of the system source code allows the system to grow and adapt much more rapidly than environments in which a formal documentation phase is a prerequisite to the development and distribution of new facilities. In turn, the users can explore the behavior of the system "all the way to the edges", as there are no sharp language barriers. The value of this linguistic uniformity has been confirmed by its successful use in other language cultures, such as Smalltalk [Goldberg, 1980].

An example: the IO system

A high level language I/O system consists of both low level device handlers and device independent sequential and random access. In most Interlisp implementations, the entire I/O system, up to and including the functions defined in the virtual machine, is provided by the host operating system. In Interlisp-D, all of the logical I/O system and a substantial proportion of the device dependent code is written in Lisp. The logical I/O system implements the Interlisp user program I/O facilities and the underlying operations in terms of which these are implemented. These include sequential and random access operations (i.e., read and write a byte, query end of file, reposition file pointer, etc.), buffer management (both for system only and directly user accessible buffers) and a device independent treatment of file properties. The logical level is in turn implemented in terms of the notion of an I/O *device*, which provides a standard set of low level, device dependent functions, such as those to read and write a page, create and delete files, etc. Thus, the addition of a new device is simply a matter of writing a new set of these functions [Kaplan *et al.*, 1980].

IMPLEMENTATION TECHNIQUES

Measurements

In tuning the performance of a program, it is crucial to be able to determine exactly where time is being spent. With a large body of code and limited manpower, it is not possible to

"optimize everything." Our performance measurement system has proved invaluable in tracking down specific (and unforeseen) problems.

The measurement system was originally developed for AltoLisp by Deutsch and Haugeland. It operates in two stages. First, the computation of interest is run with *event logging* enabled. This produces a (very large) file of *log events*, which is later analyzed. The log events are put out by both the microcode and the run time support system and include time-stamped events for function call and return, entry and exit from the Bcpl routines, I/O activity, and other events of interest. Alternatively, the microcode can also collect counts of opcode frequencies and a frequency sample of the microcode PC.

Statistics gathering can be enabled at any time that Lisp is running. One can decide spontaneously to take measurements whenever performance unexpectedly degrades. Comparison of these measurements with those taken during a similar run that exhibited normal performance can be used to identify the source of intermittent performance problems. This technique was used, for example, to track down an intermittent slowdown in the code that handled stack frame overflow.

The analysis phase reads the log file and computes summary statistics from it. From call and return events, the time spent in individual functions can be computed, either including or excluding the time spent in the functions called by them. The accumulated times (including the times spent by called functions) locate the higher level functions which are the root of a large amount of time and which may be a candidates for redesign. The individual time (excluding called functions' times) are useful for isolating what improvement can be expected from optimizing or microcoding the body of that function.

Function performance data is presented in tables which show the number of times each function was called and the time spent in each function. For example:

Function	#ofCalls	Time	%ofTime	PerCall
NTHCHC	1977	236702	10.6	119
\HT.FIND	1729	168492	7.6	97
LITLEN	2111	131708	5.9	61
LITBASE	2141	118902	5.3	56
...				

Tables such as this isolate very accurately those functions which are worth rewriting as well as identifying those which are not. In this example, NTHCHC, which calls both LITLEN and LITBASE, is an obvious candidate. In another run we discovered that 15 percent of the time was being spent adding one to a counter which had overflowed the small number range. This prompted a redesign of the large number arithmetic.

Additional controls on the analysis routines allow more specific questions to be answered. The analysis can be restricted to that part of the computation within any particular function. For example, only that part of the computation that takes place within READ can be analysed. The analysis can also be limited to a set of functions. in which case only these functions will appear in the table of results. Any time spent in a function not in the set will be charged to the closest bounding function that is.

The analysis routines extract from the log file useful information besides performance data. For example, the dynamic calling behavior is captured in the log, so one frequently useful technique is to list which functions have called (and been called by) other functions, and even how many arguments they were passed. The flexibility of the analysis routines combined with the wealth of information collected during the logging stage allows a given computation to be examined from many points of view.

Initialization

There are several areas that cause fundamental problems for the implementation of a language system in itself: memory management (which requires that the memory manager itself will not cause memory faults), stack overflow recovery (where the stack manager must itself have some stack), and initialization. Initialization is difficult because the initialization program must operate when the system is not in a well formed state. The problem in initialization can be characterized by the question: "If the compiled code reader is itself compiled code, who will read *it* in?"

Several methods of doing initialization suggest themselves. For example, the image can be initialized by a program written in some other language. This is the solution adopted in AltoLisp. Alternatively, if the interpreter is written in some other language, the compiled code reader can be run interpretively to read itself in. However, both of these solutions require a substantial amount of non-Lisp code either for storage allocation or for interpretation.

We adopted still another solution. The compiled code reader was modified to load code into an environment other than that in which it is running. The primitive functions that the loader uses to manipulate the environment (e.g., fetch and store into specified virtual memory locations) are replaced by functions that manipulate another memory image stored as a file. To begin with, an empty memory image file is created and then the "indirect" version of the compiled code reader is used to load the compiled files that constitute the lowest level of the system into this empty image. We thus avoid the potential problem of maintaining two different programs with knowledge of system data structures.

An appropriate programming environment

One of the advantages of writing most of the kernel in Lisp is that Interlisp provides a very powerful programming environment. Its attributes that we found particularly useful were:

Language features: The advantages of "data-less" or data-structure-independent programming have long been known: more readable code, fewer bugs, the ability to change data structures without having to make major source program modifications. The Interlisp record package and data type facility encourages this good practice by providing a uniform and efficient way of creating, accessing and storing data symbolically, i.e., fields of data structures are referred to by name. Because the Interlisp-D implementation allows a large number of data types, we have felt free to give system data structures (such as file-handles, page buffers, read tables) their own data types. In addition, records could be overlaid on structures not under Lisp's control (e.g., the leader page of a disk file or the format of a network packet) to provide the same uniform access.

Cross compilation: We maintained an Interlisp-10 environment in which we could edit, compile and examine functions for the Dorado. The function and record definitions for the Dorado implementation were kept on property lists instead of defintion cells. This allowed us to work on functions such as READ and CONS without destroying the environment in which we were working.

Masterscope: Many of our improvements to AltoLisp involved massive changes throughout the many system source files. Interlisp's Masterscope program was an essential aid in determining what would be affected by a proposed improvement and in actually performing the necessary edits. Masterscope is an interactive program for analyzing and cross-referencing Lisp functions. It constructs a database of which functions call which other functions, where variables are bound, used, or set, and where record declarations are referenced. Masterscope utilizes the information in the database to interpret a variety of English-like commands. Our cross-compilation environment incrementally updated a database that was shared among all programmers on the project, so that with very little overhead the information in the database was kept consistent with the current state of the evolving system.

Masterscope was most helpful in planning and carrying out modifications to major system interfaces, which usually meant changing the numbers and kinds of arguments to various functions. We would first ask Masterscope to simply list the callers of those functions to give some estimate of the impact of the proposed change, much as one might use a static cross-reference program. We would then invoke the SHOW command, instructing Masterscope to locate in the source-file definitions of all the callers the expressions that actually called the interface functions. These expressions were gathered together and displayed as a group, so that we could verify our intuitions about what assumptions clients were making about the interface. In many cases, the rapid source-code exploration that Masterscope made possible revealed flaws in our redesign which otherwise would not have become apparent until much more effort had been expended. Having decided that our modification was acceptable, we used Masterscope's EDIT command to actually drive the editing. This caused Masterscope to load the definitions of all the client functions, call the Interlisp editor on each one, and position the editor at each of the expressions that needed to be changed. Masterscope, not the programmer, kept track of which functions had been changed and which still needed to be edited. When Masterscope finished the editing sequence, the programmer was sure that the changes had been made completely and consistently.

Our redesign of the I/O system [Kaplan *et al.*, 1980] is a good illustration of the power of this interactive tool. We completely replaced the lowest-level I/O interface, which involved changes to approximately 40 functions on 15 source files. The major part of the revision was accomplished in response to a single EDIT WHERE ANY CALLS '(BIN BOUT ...) command, without ever looking at hard-copy source listings.

Rapid access to system sources: Our cross-compilation environment maintained a shared data base which allows the definition of any Lisp function to be retrieved for viewing or editing in a matter of seconds. The microcode and Bcpl could be "browsed" using the same interface. This rapid online access to the system sources greatly lessened the need to work from listings.

Levelling

One of the original motivations for having a large part of AltoLisp in Bcpl was the belief that it was important not to provide Lisp primitives that gave unrestricted access to the implementation data structures. This reasoning fails to discriminate between the system implementation and user program levels. Allowing system programs arbitrary access to memory locations does not at all imply that user level code has this access.

Failing to make the system/user distinction hurt AltoLisp in three ways. First, it provided one motivation for the large Bcpl kernel. Second, most of that part of the system which *was* written in Lisp was prohibited from manipulating underlying data structures except through overly general functional interfaces. Last, it discouraged the use of higher level structuring facilities (such as the record package) so that code that required any knowledge of system data structures tended to be written entirely in terms of low level primitives.

Using Lisp as a system implementation language requires very careful consideration of the layering of the system into levels of access and knowledge. Further, the precision that is needed cannot be obtained by simple binary discriminations but must be carefully considered for each piece of code. This presents a considerable challenge to the implementors' self restraint, as Lisp provides few facilities to enforce such a layering. Appropriate use of abstraction is essential if layering is to be preserved under the constant revision necessitated by intensive performance debugging.

Diagnostics

Development of the Lisp microcode was aided by a reasonably complete set of microcode diagnostics written in Lisp. Diagnostics are difficult because they are most useful when very little can be assumed *a priori* to work. It is also difficult to achieve complete coverage of all cases. In addition, extensive knowledge of the Lisp system was required to develop diagnostics. For example, every opcode needs to be tested when encountering page faults or stack overflows. Setting up a situation which will page fault or overflow the stack in the next opcode requires a very intimate knowledge of the implementation. Having undertaken several microcode revisions, development of a comprehensive set of diagnostics seems well worth the effort.

Important performance issues

While not strictly a technique, we feel that it is important to mention the major areas in which performance has proved to be crucial. While some of these are undoubtedly specific to Interlisp-D, we feel that they deserve consideration by those who might be building similar Lisp systems.

The earlier intuition that the hardware assist for decoding byte opcodes was important was substantiated. Performance improved by nearly a factor of two when this was installed. Implementing the decoding and dispatch in microcode is conceding a large performance loss.

There are several parts of the system for which it seems important to have microcode support. When written in Lisp, the garbage collector seems to consume between 10-30% of

the processor, although the figure varies widely over different computations. Further, in a system that uses deep binding, some form of microcode assist for free variable lookup is very desirable. A speedup factor of between two and four accompanied the introduction of microcode support for this in Interlisp-D. Statistics show that less than one percent of the execution time is now spent in free variable lookup.

Their heavy use in implementing system code almost mandates that the arithmetic functions have complete microcode support. Further, we found it to be critical to have a large range of small numbers (numbers without boxes), so that the performance critical, low level system code did not invoke Lisp's storage management.

WHY IS AN INTERLISP IMPLEMENTATION SO HARD?

The Dolphin/Dorado implementation of Interlisp took many times the expected effort to complete. Given the widespread intuition to the contrary, it is perhaps worthwhile to reflect on why it has proved so difficult. The answer is painfully simple: Interlisp is a *very* large software system and large software systems are not easy to construct. Interlisp-D has *(in June, 1980, Ed.)* on the order of 17,000 lines of Lisp code, 6,000 lines of Bcpl, and 4,000 lines of microcode. In many ways, the more interesting question is why does it look so straightforward?

Without a doubt, the perceived ease of implementing Interlisp springs from the existence of the virtual machine (VM) specification. This admirable document purports to give a complete description of the facilities that are assumed by the higher level Interlisp software, and does a remarkable job of laying out the foundations of this very large software confederation. It is difficult to resist the implication that a straightforward implementation of this mere 120 pages of specification, much of which is already described in programmatic form, will constitute a new implementation of Interlisp. The issue is rather more complicated than that.

The VM specification looks small, but it is not. There is no simple correspondence between the size of a specification and the volume of code required to implement it. Many of the major problems of an Interlisp implementation (e.g., performance, the garbage collector, the compiler) are simply not addressed at all. We caution Interlisp implementers that the slimness of that document is misleading.

Further, while the virtual machine specification is an excellent first pass, it is far from complete. Many "incidental" functions and variables were left out (e.g. HOSTNAME). It is occasionally ambiguous in places where the system code relies on a specific interpretation. Even though once complete, changes in the higher level code required that the VM be extended to support new facilities. Finding all these variations is an exhausting task. It is substantially easier to get 95% compatibility than 99.9%, and amazing how many programs are sensitive to the difference.

One way to look at the Lisp kernel that was written for Interlisp-D is as the definition of a new VM specification *in Lisp* code. While much of the code is specific to the Dorado environment, a great deal of it simply extends the virtual machine downwards by providing a much lower level treatment of functions such as PRINT and READ. We hope our work will provide other new implementations with a firmer foundation than the VM document alone.

Another problem for any very large software system is the existence of a long development tail. A version of Interlisp-D was "sort of running" years ago. Several other implementations of Interlisp have "sort of run" but have never reached production status. One of the key problems here is performance. The success of the PDP-10 implementation of Interlisp is due to a lot of hand tuning. Any straightforward, clean implementation will prove to be slow, and finding performance problems is difficult, even with good measurement tools. A large number of design decisions have to be made and a large amount of code has to be written. While not all of the decisions have to be optimal, none of them can be pessimal. While the Interlisp-D experience can provide some guidance, many of these decisions will be environment specific.

Finally, an important issue has been compatability with the PDP-10 implementation of Interlisp. In some ways our determination to remain compatible has helped. Ambiguities and omissions from the VM specification could always be resolved by copying the PDP-10 implementation. However, this compatibility requirement was also a burden. Complete compatibility with another implementation is hard. This is particularly so when the new implementation is in a quite different environment (a personal rather than a time-shared machine). The tension between remaining compatible versus exploring the possibilities of a personal machine environment is a continuing issue, which will probably be a focus of our further efforts on the Interlisp-D system.

Acknowledgements

Peter Deutsch was a principal designer and motivating force behind AltoLisp, of which Interlisp-D is a successor. Warren Teitelman has made major contributions to the Interlisp-D project. Martin Kay, Henry Thompson, Richard Fikes and Austin Henderson have also contributed time and effort on various aspects of the project.

REFERENCES

Bobrow, D.G. & Clark, D.W.
Compact encodings of list structure. *ACM Transactions on programming languages and systems* **1**, 1979.

Deutsch, L.P.
A Lisp machine with very compact programs. *Proceedings of the third international joint conference on artificial intelligence,* Stanford 1973.

Experience with a microprogrammed Interlisp system. *IEEE Micro-11 conference*, 1978.

Deutsch, L.P. & Bobrow, D.G.
An efficient incremental, automatic garbage collector. *CACM* **19**:9, 1976.

Fiala, E.R.
The Maxc systems. *IEEE Computer* **11**, May 1978.

Goldberg, A.
Smalltalk: Dreams and schemes. Xerox PARC, to appear.

Kaplan, R.M., Sheil, B.A., & Burton, R.R.
The Interlisp-D I/O system. Xerox PARC, SSL-80-4, 1980.

Lampson, B.W. & Pier, K.A.
A processor for a high-peformance personal computer. *Seventh international symposium on computer architecture*, La Baule, France, May 1980.

Masinter, L.M. & Deutsch, L.P.
Local optimization in a compiler for stack-based Lisp machines. *Proceedings of the 1980 Lisp conference*, Stanford, 1980 and Xerox PARC, SSL-80-4, 1980.

Moore, J.S.
The Interlisp *virtual machine specification.* Xerox PARC, CSL-76-5, 1976.

Teitelman, W. *et al.*
Interlisp *Reference Manual*, Xerox PARC, 1978.

Interlisp-D: Further steps in the flight from time-sharing

B. A. Sheil

Abstract

The Interlisp-D project was formed to develop a personal machine implementation of the Interlisp programming environment to support research in artificial intelligence and cognitive science [Burton *et al.*, 80b]. This note describes the principal developments since our last report [Burton *et al.*, 80a].

INTERLISP-D

Interlisp-D is an implementation of the Interlisp programming environment [Teitelman & Masinter, 81] for the Dolphin and Dorado personal computers. Both the Dolphin and Dorado are microprogrammed personal computers, with 16-bit data paths and relatively large main memories (~1 megabyte) and virtual address spaces (4M-16M 16 bit words). Both machines have a medium sized local disk, Ethernet controller, a large raster scanned display and a standard Alto keyboard and "mouse" pointing device.

Both the internal structure of Interlisp-D and an account of its development are presented in [Burton *et al*, 80b]. Briefly, Interlisp-D uses a byte-coded instruction set, deep binding, CDR encoding (in a 32 bit CONS cell) and incremental, reference counted garbage collection. The use of deep binding, together with a complete implementation of spaghetti stacks, allows very rapid context switching for both system and user processes. Virtually all of the Interlisp-D system is written in Lisp. A relatively small amount of microcode implements the Interlisp-D instruction set and provides support for a small set of other performance critical operations. The at one time quite large Bcpl kernel has been all but completely absorbed into Lisp, for the reasons outlined in [Burton *et al*, 80b].

Interlisp-D is completely upward compatible with the widely used PDP-10 version. All the Interlisp system software documented in the Interlisp Reference Manual [Teitelman *et al.*, 78] runs under Interlisp-D, excepting only a few capabilities explicitly indicated in that manual as applicable only to Interlisp-10. The completeness of the implementation has been demonstrated by the fact that several very large, independently developed, application systems, such as the KLONE knowledge representation language [Brachman, 78], have been brought up in Interlisp-D with little or no modification. Interlisp-D is in active use by researchers (other than its implementors) at several different research sites and is now approaching the level of stability and reliability of Interlisp-10.

CURRENT PERFORMANCE

The performance engineering of a large Lisp system is distinctly non-trivial. We have invested considerable effort, including the development of several performance analysis tools, on the performance of Interlisp-D and, as a result, seen its performance improve by nearly a factor of five over the last year. Although relative performance estimates can be misleading, because of variation due to choice of benchmarks and compilation strategy, the

A revised version of a paper originally published in SIGART Newsletter, No. 77, August 1981.

overall performance of Interlisp-D on the Dolphin currently *(in June 1981, Ed.)* seems to be about twice that of Interlisp-10 on an otherwise unloaded PDP KA-10. Although this level of performance makes the Dolphin a comfortable personal working environment, we have already identified a number of improvements which we anticipate will further improve execution speed by between 20% and 100%. Furthermore, we believe that we are nowhere near the point of diminishing returns for this kind of performance engineering.

MACHINE INDEPENDENCE

Another major thrust has been to reduce the dependencies on specific features of the present environment, so as to facilitate Interlisp-D's implementation on other hardware. Dependencies on the operating system have been removed by absorbing most of the higher (generally machine independent) facilities provided by the operating system into Lisp code. Gratuitous dependencies on attributes of the hardware, such as the 16-bit word size, have been removed and inherent ones isolated. In addition to an abstract desire for transportability, our sharing of code with other Interlisp implementation projects provides an on-going motivation for this effort.

EXTENDED FUNCTIONALITY

The principal innovations in Interlisp-D, with respect to previous implementations of Interlisp, involve the extensions required to allow the Interlisp user access to a personal machine computing environment.

Network facilities

While network access is a valuable facility in any computing environment, it is of particular importance to the user of a personal machine, as it is the means by which the shared resources of the community are accessed. Over the last year, Interlisp-D has incorporated both low level Ethernet access and a collection of various higher level protocols used to communicate with the printing and file servers in use at PARC. It is now straightforward to conduct *all* file operations directly with remote file servers. This both allows the sharing of common files (e.g., for multi-person projects, such as the construction of Interlisp-D itself), permits a user to move easily from one machine to another, and eliminates any constraints of local disk size. We have also begun to investigate the possibility of paging from a remote virtual memory elsewhere on the network. This would not only allow completely transparent relocation of a user's environment from one machine to another, but would open up a variety of interesting schemes for distributing a computation across a set of machines.

High level graphics facilities

Interlisp-D has always had a complete set of raster scan graphics operations (documented in [Burton, 80b]). More recent developments include a collection of higher level user graphics facilities, akin to those found in other personal computing environments. The most important of these is the Interlisp-D window package. This facility differs in spirit from most other window systems in that, rather than imposing an elaborate structure on programs that use it, it is a self consciously *minimal* collection of facilities which allow multiple programs to share the same display. Although some mechanism is necessary to adjudicate a harmonious sharing of the display, we feel that higher level display structuring conventions are still an

open research question and therefore should not yet be incorporated into a mandatory system facility. The window package *does* provide both interactive and programatic constructs for creating, moving, reshaping, overlapping and destroying windows, in such a way that a program can be embedded in a window in a completely transparent (to that program) fashion. This allows existing programs to continue to be used without change, while providing a base for experimentation with more complex window semantics in the context of individual applications.

One such existing application is the display based, structural program editor. This editor, in contrast to the character orientation of most modern display based program editors, is the result of marrying display techniques (selection and command specification by pointing, incremental reprinting, *etc*) with the structure orientation of the existing Interlisp editor. Indeed, the two editors are interfaced so that the considerable symbolic editing power of the existing editor remains available under the display based one. Although our initial experience has been positive, the user interface is under continuing revision as we gain further experience with this style of editing.

FUTURE PLANS

The area in which we anticipate most future development of Interlisp-D is the personal computing facilities, such as graphics and networking, and their integration into Interlisp's rich collection of programming support tools. While radical changes to the underlying language structures are made difficult by our desire to preserve exact Interlisp compatibility, we also expect some language extensions, including some form of object oriented procedure invocation.

One of the great strengths of Interlisp has been the many contributions made by its active, critical user community. We are hopeful that the recent commercial availability of Interlisp-D to other sites, and the consequent growth of its user community, will be a similar source of long term strength and, in the short term, significantly accelerate the pace with which Interlisp evolves away from its time-shared origins into a personal computing environment.

REFERENCES

Brachman, R. *et al.*
KLONE Reference Manual. BBN Report No. 3848, 1978.

Burton, R. *et al.*
Overview and status of DoradoLisp. *Proceedings of the 1980 Lisp Conference*, Stanford, 1980a.

Burton, R. *et al.*
Papers on Interlisp-D. Xerox PARC report, SSL-80-4, 1980b.

Teitelman, W. *et al.*
The Interlisp reference manual. Xerox PARC, 1978.

Teitelman, W. and Masinter, L.
The Interlisp programming environment. *IEEE Computer*, 14:4 April 1981, pp. 25-34.

Local Optimization in a Compiler for Stack-based Lisp Machines

Larry M. Masinter and L. Peter Deutsch

Abstract

We describe the local optimization phase of a compiler for translating the Interlisp dialect of Lisp into stack-architecture (0-address) instruction sets. We discuss the general organization of the compiler, and then describe the set of optimization techniques found most useful, based on empirical results gathered by compiling a large set of programs. The compiler and optimization phase are machine independent, in that they generate a stream of instructions for an abstract stack machine, which an assembler subsequently turns into the actual machine instructions. The compiler has been in successful use for several years, producing code for two different instruction sets.

INTRODUCTION

This paper describes the local optimization phase of a compiler for translating the Interlisp dialect of Lisp [Teitelman *et al.*, 1978] into stack-architecture (0-address) instruction sets [Deutsch, 1973]. We discuss the general organization of the compiler, and then the set of optimization techniques we have found most useful. The compiler and optimization phase are machine independent, in that they generate a stream of instructions for an abstract stack machine, which an assembler subsequently turns into the actual machine instructions. The compiler has been used for several years, producing code both for an 8-bit Lisp instruction set for several personal computers [Deutsch, 1978, 1980; Burton *et al.*, 1980], and a 9-bit instruction set for Maxc, a time-shared machine running the Tenex operating sytem [Fiala, 1978].

There are always tradeoffs in designing a compiler. Each additional optimization usually increases the running time of the compiler as well as its complexity. The improvement in the code generated must be weighed against the benefit gained, measured by the amount of code improvement weighted by the frequency with which the optimization is applicable. Rather than provide a multiplicity of compiler controls, which most users would not want to know about, the compiler designer should use empirical knowledge of "average" user programs and make appropriate design choices. One of the major purposes of this paper is to publish some empirical results on the relative utility of different code transformations, which can aid designers in making such choices.

Why this compiler is different

Compiling Lisp for a 0-address architecture differs from compiling other languages such as PASCAL or ALGOL for several reasons. Procedures are independently compiled, so that global optimization techniques are not relevant. Compiling for a stack-based instruction set is different from compiling for more conventional machine architectures, in that register allocation is not relevant, and randomly addressable compiler-generated temporary variables other than top-of-stack are difficult to access.

A revised version of a paper originally presented at the 1980 Lisp Conference, Stanford, Ca.

In systems which provide interactive, symbolic debugging of compiled code, a compiler must not manipulate source programs too freely, since even common optimizations like tail recursion removal make it difficult or impossible to explain the dynamic state of the program in terms of the original source. However, Lisp also provides an interpreter which can be used for debugging purposes when strict faithfulness is needed; interpreted and compiled code can be mixed freely. Thus, we take the view that the compiler can rearrange the implementation of an individual function in any manner consistent with the semantics of the original program, even if fine-grained debugging information may be lost or altered (e.g., if variables that appeared in the source get eliminated).

What we did not handle

The compiler concentrates on local optimizations. More global transformations such as pulling invariants out of loops or duplicate expression elimination would probably pay off often enough to be worth the additional complication in an environment where speed was of great concern and the individual functions were large.

Related work

A few of our compiler's transformations, such as cross jumping and tail recursion removal, have been part of the literature for some time. We know of three other Lisp compilers that both compile into a machine-independent intermediate language and do substantial optimization.

The Standard Lisp project at the University of Utah has produced a transportable compiler similar to ours [Griss & Hearn, 1979]. Their intermediate language is register, rather than stack, oriented. Their report mentions a number of the optimizations in our list, plus others only applicable to register machines, but their list is shorter and not accompanied by empirical data.

Another similar compiler was the subject of a Ph.D. dissertation [Urmi, 1978]. The author in this case was more concerned with the design of instruction sets than with optimizing the use of a given architecture. His report contains extensive statistics on the opcode frequencies, and interesting suggestions for instruction set design, including a consideration of both stack- and direct-address architectures; however, his optimizations are all in the "peephole" category, being limited to a few adjacent instructions, except for the usual optimization of ANDs and ORs.

The RABBIT compiler [Steele, 1978] translates an unusual lexically scoped Lisp dialect into code for a register machine. Its optimization techniques are extremely sophisticated with regard to removal of recursions and variable bindings. However, the differences in coding style resulting from lexical scoping are so large that a comparison between RABBIT's goals and those of our compiler would not be meaningful.

Results

Optimization in the byte compiler provides an average 5-10% speed improvement and a 10-15% space improvement over completely unoptimized code. While significant, this does not make it one of the more significant factors affecting the performance of our Lisp systems

[Burton *et al.*, 1980]. The most significant effect that a reasonable optimizing compiler has for its users is a certain amount of unconcern for vagaries of syntax. Programmers can write their routines for clarity, without concern for purely syntactic devices which might otherwise affect performance. For example, while inserting assignments inside expressions is allowed and occasionally perspicuous, it generally is more readable to perform variable assignments in separate statements, and to subsequently use the variables in an unnested manner. Knowing that the compiler will do an adequate job of optimization means that a program author can make choices based on legibility, even in the most time-critical routines.

ABOUT THE COMPILER AND THE OBJECT LANGUAGE

The compiler operates in several passes. The first pass takes the S-expression definition of the function being compiled, and walks down it recursively, generating a simple intermediate code, called ByteLap, analogous to assembly code. During this first pass, the compiler expands all macros, CLISP, record accesses and iterative statements. A few optimizations are performed during this pass, but most of the optimization work is saved for later. The next pass of the compiler is a "post-optimization" phase, which performs transformations on the ByteLap to improve it. Transformations are tried repeatedly, until no further improvement is possible.

After the post-optimization phase is done, the results are passed to an assembler, which transforms the ByteLap into the actual machine instructions. We currently have two different assemblers in use, which generate code for two different instruction sets: one for the Maxc 9-bit instruction set and one for the personal machine 8-bit instruction set. The Maxc and personal machine implementations of Interlisp differ considerably; for example, the Maxc system employs shallow variable binding, while the personal machine systems employ deep binding. The translation from ByteLap to machine code is straightforward.

The structure of ByteLap

The ByteLap intermediate code generated by the compiler can be viewed as the instruction set for an abstract stack machine. The format of ByteLap is described here to simplify subsequent discussion of optimizations. There are 15 opcodes, each of which has some effect on the state of the linear temporary value stack. The instruction set is:

(VAR **var**)	Push the value of the variable **var** on the stack.
(SETQ **var**)	Store the top of the stack into the variable **var**.
(POP)	Pop the stack (i.e., throw away the top value and decrement stack depth by one).
(COPY)	Duplicate (push again) the top of the stack.
(CONST **val**)	Push the constant **val** on the stack (**val** may be of any Lisp data type, e.g., an atom or a number.)
(JUMP **tag**)	Jump to the location **tag**.

(FJUMP **tag**)	Jump to the indicated location if top-of-stack is NIL, otherwise continue. In either case, pop the stack.
(TJUMP **tag**)	Similar to FJUMP, but jump if top-of-stack is non-NIL.
(NTJUMP **tag**)	Similar to TJUMP, but do not pop if it jumps. This is useful when a value is tested and then subsequently used.
(NFJUMP **tag**)	Analogous to NTJUMP.
(FN **n fn**)	Call the function **fn** with **n** arguments.
(BIND ($v_1 \ldots v_n$)($n_1 \ldots n_k$))	Bind the variables $v_1,\ldots,v_n$ to the **n** values on the top of the stack. Also bind the variables $n_1,\ldots,n_k$ to NIL. All bindings are done in parallel. Remember the current stack location.
(UNBIND)	Save the current top of stack. Throw away any other values on the stack since the last (stacked) BIND, and undo the bindings of that BIND. Re-push on the stack the saved value. This is used at the end of PROG or LAMBDA expressions whose value is used.
(DUNBIND)	Similar to UNBIND, but do not restore the value.
(RETURN)	Return top-of-stack as the value of the current function, throwing away any other values on the stack.

Note that a given ByteLap opcode could have one of several different translations in the actual code executed. For example, both the personal machine and Maxc implementations have a separate opcode for pushing NIL, in addition to a more general constant opcode. The final code generation phase transforms the (CONST NIL) ByteLap instruction into the appropriate opcode. Operations such as arithmetic or CAR are encoded as FN calls, even though the instruction sets have specialized instructions to perform those operations. The assemblers distinguish between the built-in operations and those that must actually perform external calls; the compiler and the optimization phase do not care. Furthermore, a sequence of ByteLap instructions can assemble into a single machine instruction; for example, both instruction sets have instructions which can do a SETQ and a POP in the same instruction. These are easily detected with a short look-ahead during code generation.

COMPILER OPTIMIZATIONS

One of the most important ground rules for the optimization phase has been that all optimizations are conservative: they must not increase either code size or running time. Only optimizations which experience has shown to be useful are described here.

The statistics given in the text below were obtained as a result of compiling a total of about 2200 functions, producing 65000 bytes of object code. Numbers in <angle brackets> indicate the number of times that a given optimizing transformation or technique was applicable.

Optimizations during code generation

A few optimizations are performed during the initial code generation phase. In particular, the

compiler keeps track of the execution context of any given expression (similar to many other Lisp compilers we know of). Thus, in the recursive descent of the S-expression definition, the flag **effect** is set if the current expression is being compiled for effect only, and the flag **return** if the value is being returned as the value of the entire function.

Remove no-effect constructs when compiling for effect <162>

Compiling a variable or constant for effect results in no code generated. A call to a function with no side effects merely causes its arguments to be evaluated for effect: for example, a macro might expand into (CAR (RPLACA X Y)), which if executed for effect only performs the RPLACA, but if the value is used will return the value stored.

Remove extraneous POP <2035>

Knowledge of **return** context is used to omit extraneous POP instructions, since unused values can be left on the stack to be swept away when the frame is released by a (RETURN). For example, in the function

```
(LAMBDA (X) (PRINT X) (TERPRI))
```

the first pass emits

```
(VAR X) (FN 1 PRINT) (FN 0 TERPRI) (RETURN)
```

rather than

```
(VAR X) (FN 1 PRINT) (POP) (FN 0 TERPRI) (RETURN).
```

The compiler also uses **return** context to eliminate extraneous JUMPs after arms of a conditional to the end of the conditional code (each arm of the conditional is compiled in **return** context, which will cause it to be terminated by a (RETURN) opcode).

The compiler also removes tail recursion in **return** context <36>. In addition, constant folding is done in the first pass for functions which are constant on constant arguments (e.g. EQ and arithmetic opcodes) <34>. Constant folding is done after the code for each argument is generated, so that constant detection can be achieved by looking for CONST opcodes, rather than pre-expansion of macros.

Post-optimizations

The second pass of the compiler consists of several local transformations on the generated ByteLap code which are tried repeatedly in turn until no further improvement can be made <6461 passes total, including the final unsuccessful pass on each function>. While the compiler contains many transformations, empirical results of compiling a large number of files show that the following transformations are the most useful—we have excluded transformations which were rarely effective. For each transformation we give its name, a symbolic version of it, a brief discussion, and an example in which the optimization would be effective.

COPY introduction <1018>

val val ⇒ **val** (COPY)

This transformation reduces neither code size nor execution time; however, it often enables

other optimizations. The **val** opcodes can be two identical CONST or VAR opcodes, or a SETQ followed by a VAR with the same variable. For example, the expression

```
(FOO (SETQ X (FUM)) X)
```

compiles to

```
(FN 0 FUM) (SETQ X) (VAR X) (FN 2 FOO)
```

which gets transformed to

```
(FN 0 FUM) (SETQ X) (COPY) (FN 2 FOO).
```

Variable duplication <1137>

(SETQ **var**) (POP) (VAR **var**) ⇒ (SETQ **var**)

This transformation occurs frequently after assignments. For example, the expressions

```
(SETQ X Y) (COND (X (FN)))
```

compiles to

```
(VAR Y) (SETQ X) (POP) (VAR X) (TJUMP L1) (FN 0 FN) L1:
```

which transforms into

```
(VAR Y) (SETQ X) (TJUMP L1) (FN 0 FN) L1:
```

Dead assignment <661>

(SETQ **var**) *{no subsequent use of* **var***}* ⇒

The compiler scans ahead a short distance for either a (RETURN) or subsequent (SETQ **var**) with no intervening instruction which either uses (VAR **var**) or else calls a function which might see the binding of **var**. For example, after the examples in both *COPY introduction* and *Variable duplication*, the assignment to X might well be "dead", and the (SETQ X) removed.

Unused push <734>

val (POP) ⇒

Although the first pass avoids generating values followed by POP by the **effect** mechanism, enough instances arise where subsequent optimizations uncover unused values to make this transformation worthwhile during the post-optimization phase. **val** can be a CONST, VAR, or COPY. In addition, if **val** is a (FN **n fn**), where **fn** is a side-effect free function, it is replaced by **n** (POP)s.

Merge POP with DUNBIND <105>

(POP) (DUNBIND) ⇒ (DUNBIND)

This simple transformation takes advantage of the fact that the DUNBIND opcode implicitly pops any values left on the stack since the last BIND.

JUMP OPTIMIZATIONS

Vacuous jump <1033>

(JUMP **tag**) **tag:** ⇒

(cJUMP tag) tag: ⇒ (POP)

While the first pass ByteLap generation explicitly deletes these <265 occurrences>, this transformation is useful to clean up after others. In the pattern, cJUMP is either TJUMP or FJUMP.

Invert sense of jump <488>

(FJUMP tag1) (JUMP tag2) tag1: ⇒ (TJUMP tag2)

This transformation can occur, for example, when there are explicit GO's in the source. For example, the expression

```
(COND (X (GO LABEL1)))
```

compiles to

```
(VAR X) (FJUMP L1) (JUMP LABEL1) L1:
```

which transforms into

```
(VAR X) (TJUMP LABEL1) L1:
```

COPY introduction for TJUMP <241>

val (NTJUMP tag) val ⇒ val (COPY) (TJUMP tag)

This transformation notes that, whether or not the JUMP is taken, the value **val** will remain on the stack. The transformation is effective for both NTJUMP and NFJUMP. Note that **val** will be NIL in one of the cases.

JUMP code in-line <457>

(JUMP tag) ... tag: *{code}* **⇒** *{code}* **...**

This transformation moves the entire segment *{code}* in line only in the situation where the JUMP is the only way of reaching **tag**.

Jump-through <2259>

(jump tag) ... tag: (JUMP tag2) ⇒ (jump tag2) ...

One of the most common transformations in the compiler occurs when the target of a jump is itself a jump instruction. For example, the code generated for

```
(COND (A B) (T C))
```

is:

```
(VAR A) (FJUMP L1) (VAR B) (JUMP L2) L1: (VAR C) L2:
```

If the variable B is replaced by a COND clause, the target of the jump at the end of that CONDs second clause would itself be a jump instruction. The **jump** in the pattern above can be any of the four jump opcodes. For example,

```
(COND (A B) (T (GO TAG))
```

would result in the fragment:

```
(VAR A) (FJUMP L2) ... L2: (JUMP TAG)
```

which can be transformed into

```
(VAR A) (FJUMP TAG) ...
```

Unreachable code <1670, removed 1784 instructions>

```
(JUMP tag) {code} ⇒ (JUMP tag)
```

The code after a JUMP or RETURN which is not itself jumped to can be deleted. The first pass avoids generating any constructs of this form, but such situations can be generated by other transformations. For example, in both preceding examples, the code at L2 might well be unreachable and deleted.

NTJUMP introduction <610>

```
val (TJUMP tag) ... tag: val ⇒ val (NTJUMP tag+1) ...
```

This optimization is essentially *COPY introduction* across jumps. For example,

```
(PROG NIL LP (FOO X) (COND ((SETQ X (CDR X)) (GO LP))) ...)
```

results in

```
LP: (VAR X) (FN 1 FOO) (POP) (VAR X) (FN 1 CDR) (SETQ X)(TJUMP LP) ...
```

which is then transformed to

```
(VAR X) LP1: (FN 1 FOO) (POP) (VAR X) (FN 1 CDR) (SETQ X) (NTJUMP LP1)
```

NTJUMP introduction with code movement <506>

```
 val (FJUMP tag) val {code1} ... tag: {code2}
⇒ val (NTJUMP tag2) {code2} tag2: {code1}
```

This transformation is a variation of *NTJUMP introduction* where it is necessary to move code around. The two code sequences *{code1}* and *{code2}* must end with a JUMP or a RETURN. Note that this transformation moves the entire segment of code *{code2}* inline. For example, the expressions

```
(COND (X (FN1 X)) (T (FN2) (GO LAB))
```

compile to

```
(VAR X) (FJUMP L1) (VAR X) (FN 1 FN1) (JUMP L2)
L1: (FN 0 FN2) (JUMP LAB) L2:
```

which gets transformed to

```
(VAR X) (NTJUMP L3) (FN 0 FN2) (JUMP LAB) L3: (FN 1 FN1) (JUMP L2) L2:
```

Jump to NIL/POP <834>

```
(FJUMP tag) ... tag: (CONST NIL) ⇒ (NFJUMP tag+1)
(NcJUMP tag) ... tag: (POP) ⇒ (cJUMP tag+1)
```

The pattern NcJUMP stand for either flavor of N-conditional jump. In the first situation, the NIL which is being found by the FJUMP may be logically distinct from the NIL after tag. For example, the expression

```
(COND (A ...) (T (MYFN NIL))
```

compiles as

```
(VAR A) (FJUMP L1) ... L1: (CONST NIL) (FN 1 MYFN)
```

which is transformed into

```
(VAR A) (NFJUMP L2) ... L2: (FN 1 MYFN).
```

The second form normally occurs only after other transformations, where a conditional, originally thought to be executed for value, does not need the value being preserved.

Removal of loop variables <679>

```
    (SETQ var) (POP) (JUMP tag) ... tag: (VAR var)
 ⇒ (SETQ var) (JUMP tag+1)
```

This transformation is common in loops. For example,

```
(PROG NIL LP (PROCESS X) (SETQ X (NEXT X)) (GO LP))
```

compiles as

```
LP: (VAR X) (FN 1 PROCESS) (POP) (VAR X) (FN 1 NEXT) (SETQ X) (POP)
    (JUMP LP)
```

This transforms to:

```
LP: (VAR X) LP1: (FN 1 PROCESS) (POP) (VAR X) (FN 1 NEXT) (SETQ X)
    (JUMP LP1)
```

Cross jumping <1721>

{code} (JUMP **tag**) ... *{code}* **tag:** ⇒ (JUMP **tag2**) ... **tag2:** *{code}*

This frequent transformation improves code space with no effect on running time. For example, the expression

```
(COND (A (FOO X)) (T (FOO Y))
```

compiles as

```
(VAR A) (FJUMP L1) (VAR X) (FN 1 FOO) (JUMP L2)
L1: (VAR Y) (FN 1 FOO) L2:
```

The instruction before (JUMP L2) is identical to the instruction before the label L2, and so this can be transformed into

```
(VAR A) (FJUMP L1) (VAR X) (JUMP L3) L1: (VAR Y) L3: (FN 1 FOO)
```

Jump copy test <733>

val fn1 (jump tag) val ... tag: val ⇒ val (COPY) **fn1 (jump tag+1)**

In this transformation, **fn1** is a "clean" function of one argument, e.g., (FN 1 LISTP) or (FN 1 CDR), or even (CONST **val**) (FN 2 EQ). In this case, "clean" means that the function cannot change the value of **val**. For example, the expression:

```
(COND ((LISTP X) (CAR X)) ((NUMBERP X) (ADD1 X)))
```

results in the fragments

```
(VAR X) (FN 1 LISTP) (FJUMP L1) (VAR X) ... L1: (VAR X) (FN 1 NUMBERP)...
```

which transforms into

```
(VAR X) (COPY) (FN 1 LISTP) (FJUMP L2) ... L2: (FN 1 NUMBERP)...
```

Return optimizations

Return merge

```
    (TJUMP tag) {code} (RETURN) ...tag2: {code} (RETURN)
 ⇒ (FJUMP tag2) ... tag2: {code} (RETURN)
```

This is an effective code transformation which can merge completely unrelated (with regard to flow-of-control) return sequences. It does not affect speed, only space. *Return merge* is unique in not preserving the normal invarient that stack-depth is constant at any location in

the code. Normal code generation only creates sequences of instructions where the stack-depth at any location is static; all other transformations preserve that property. However, the two occurrences of *{code}* in the pattern need not be at the same stack-depth, and thus, stack-depth would be ambiguous after **tag2**. This is important if the target machine language is dependent upon stack depth in the translation from ByteLap, as is the case with the Maxc instruction set. *Return merging* must be disabled if the two *{code}* sequences occur at different stack depths, and if *{code}* contains any stack-level-sensitive operations.

Needless POP before RETURN <590>

(POP) **val** (RETURN) ⇒ **val** (RETURN)

This transformation is attempted only after it is known that there is no opportunity for *Unused push*. In addition to removing POP opcodes, this transformation also removes DUNBIND and UNBIND opcodes in the same position (except when **val** is a variable which was bound in the frame corresponding to the UNBIND or DUNBIND).

Unused variable in BIND <580>

(BIND ... (.. **var** ..)) *{***var** *not used}* ⇒ (BIND ... (.. ..))
(BIND (.. **var**) ...) *{***var** *not used}* ⇒ (POP) (BIND (..) ...)

This transformation eliminates binds of local variables which are not used. Only the last variable bound to a value can be so removed, because of the difficulty of inserting a POP at the appropriate place back in the instruction stream. (This is an example where source level transformation might be better way of doing optimization. Unfortunately, the last use of a variable is often removed by *COPY introduction*, which has no analogue in source code transformations.) To detect unused variables, the compiler scans the code linearly for uses of each variable in every BIND. For example, the expression

(PROG (X) (SETQ X (FUM)) (FOO X X))

compiles into

(BIND () (X)) (FN 0 FUM) (SETQ X) (POP) (VAR X) (VAR X) (FN 2 FOO)

which, after several transformations, turns into

(BIND () (X)) (FN 0 FUM) (COPY) (FN 2 FOO).

Since X is no longer used, it can be eliminated. Note that this transformation is not applicable to special variables (variables which can be referenced freely by functions called from this one, e.g., FUM and FOO).

Unused BIND <2035>

(BIND (**v1** ... **vm**) (**vm+1** ... **vn**)) (VAR **v1**) ... (VAR **vm**) *{last mention of* **v1**...**vm***}*
⇒ (CONST NIL) *{**n-m** times}*

<Of the 2035 occurrences, 440 eliminated BINDs which were generated in the compilation of mapping functions.> This transformation eliminates BINDs when the variable list is empty or when the variables bound are only mentioned, in order, immediately following the BIND. When this transformation is made, the compiler must also find all corresponding DUNBIND's for this frame and turn them into the appropriate number of POP's. In addition, for every UNBIND the stack level must be exactly one greater than it was at the BIND. If so, the UNBIND can simply be deleted; if not, this transformation cannot be made. Note, however,

that where a PROG or LAMBDA expression is the value returned by a function, no UNBIND or DUNBIND opcodes are generated. For example, the expression

```
((LAMBDA (X) (FOO X X)) (FUM))
```

compiles into

```
(FN 0 FUM) (BIND (X) ()) (VAR X) (VAR X) (FN 2 FOO)
```

which, after *COPY introduction* and *Unused BIND* can be transformed into

```
(FN 0 FUM) (COPY) (FN 2 FOO).
```

CONCLUSIONS

Because our instruction sets are so well suited to the Lisp language, it is possible to write quite simple non-optimizing compilers for our Lisp machines. In fact, we have written a simple but usable compiler in less than three pages of Lisp code. However, local transformations can have an important impact on code space and running time.

As in production systems, the choice of order of application of transformations can affect the results. Without effectively trying all possible orderings, one transformation can prevent a better one from being used. In successive transformations made on a sample of user Lisp programs, however, we have not observed this to be a major problem.

The programs our compiler generates are still not optimized, in the strict sense of that term. A sample of user Lisp programs which were "hand optimized" show that code size could be compressed by as much as an additional 15% in some cases, with no speed penalty. However, the transformations involved seem to require either much special-case pattern matching or else transformations which temporarily reduce either space or speed. As usual when employing "hill-climbing" algorithms, by requiring that all transformations we employ are strict improvements, we occasionally find local optima which prevent better solutions from being found.

Optimizing on a simple intermediate language is quite effective. Many of the transformations made are not expressible as source language transformations (e.g., the COPY operator has no direct counterpart in the Lisp language). Those that would be easier to express as source transformations are often enabled by transformations which have no direct analogue. Peephole optimizers working on more complex assembly languages must be aware of more special cases, because there are many more kinds of operations.

REFERENCES

Burton, R.R. *et al.*
Overview and implementation status of Interlisp-D. *Proceedings of the 1980 Lisp conference*, Stanford, 1980 and Xerox PARC, SSL-80-4, 1980.

Deutsch, L.P.
A Lisp machine with very compact programs. *Proceedings of the third international joint conference on artificial intelligence*, Stanford 1973.

Experience with a microprogrammed Interlisp system. *IEEE Micro-11 conference*, 1978.

ByteLisp and its Alto implementation. Proceedings of the *1980 Lisp Conference*, Stanford, 1980.

Fiala, E.R.
The Maxc systems. *IEEE Computer* 11, May 1978.

Griss, M.L. & Hearn, A.C.
A portable Lisp compiler. *Department of Computer Science, University of Utah, UCP-76,* 1979.

Steele, G.L.
RABBIT: A compiler for SCHEME (A study in compiler optimization). *MIT Artificial Intelligence Laboratory, AI-TR-474,* 1978.

Teitelman, W. *et al.*
Interlisp *Reference Manual*, Xerox PARC, 1978.

Urmi, Jaak
A machine independent Lisp compiler and its implications for ideal hardware. *Linkoping studies in science and technology dissertations No. 22*, Linkoping, Sweden, 1978.

XEROX

Papers on Interlisp-D

B.A. Sheil and Larry M. Masinter, editors

Cognitive and Instructional Sciences Series CIS-5 (Revised)