Integration, extensibility, and ease of modification made Interlisp unique and powerful. Its adaptations will enhance the power of the coming world of personal computing and advanced displays.

The Interlisp Programming Environment

Warren Teitelman and Larry Masinter
Xerox Palo Alto Research Center

Interlisp is a programming environment based on the Lisp programming language.[1,2] In widespread use in the artificial intelligence community, Interlisp has 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 now has approximately 300 users at 20 different sites (mostly universities) in the US and abroad. It is an extremely well documented and well maintained system.

Interlisp has been used to develop and implement a wide variety of large application systems. Examples include the Mycin system for infectious disease diagnosis,[3] the Boyer-Moore theorem prover,[4] and the BBN speech understanding system.[5]

This article describes the Interlisp environment, the facilities available in it, and some of the reasons why Interlisp developed as it has.

Overview

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 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."[6]

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 applications 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 the course of 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 structured 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

*Sandewall's excellent survey article, "Programming in an Interactive Environment: The Lisp Experience," gives an overview of existing programming methodology in the Lisp user's 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.

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.''[7]

Computer costs vs. people costs. The second major influence in Interlisp's development was a willingness to "let the machine do it." The developers 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 in respect to the available computer resources.

The Advanced Research Projects Administration of the Department of Defense sponsored much of this effort. As a result, we had a fair amount of freedom in the development of Interlisp, i.e., we did not have to justify the cost-effectiveness of our research to a profit-oriented management. ARPA's 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 is 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 the 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 tools and 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 Interlisp facilities

File package. Interactive program development consists alternately of testing of program parts and editing to correct errors discovered during the tests and/or extend the program. Interlisp, unlike many other interactive programming systems, supports both the 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.[7]

In a residential system, it is important to be able to take procedures represented by data structures and print them on text files in an input-compatible format for use as backup, for transporting programs from one environment to another, and to provide hard-copy listings. In Interlisp, the file package is a set of functions, conventions, and interfaces with other system packages. The role of the file package is to automate the bookkeeping necessary for 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 a particular 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.[8]

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 significant breakthrough occurred with the emergence of the idea of having the system notice when a datum was changed and associate this fact with the file containing the datum.

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 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 the system to 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 of the idea (probably through some user saying, "Wouldn't it be nice if the system . . .") 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, operated by discovering which file(s) contained the datum and associating with each such file the name of the object that had changed. Calls to MarkAsChanged were then inserted in all of the parts of the Interlisp system that changed objects—the editor, the DEFINE function, the facility for (re)declaring records, and DWIM (which can modify a function when it performs a spelling correction or some other transformation inside the function).

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 that contain data 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 prompt the user about unfiled objects and to allow the user to specify the destinations of these objects.

By this point, the Interlisp user did not have to worry about maintaining his source files, save for occasionally calling Cleanup. The file package, however, 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 directly 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 a session, the user manipulates the pieces with a variety of tools, occasionally saving what he has done by calling the Cleanup function. 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 between 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, like the file package, has its roots in an extremely simple program.

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 program 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 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 reanalysis, 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 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. According to Sandewall, "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."[7]

The most visible part of DWIM[9] 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 facilties 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 from 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 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: (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 approx imately 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.[10] 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,[8] 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 re-execute the modified operations, the USE command performs a substitution before re-executing a specified event (e.g., USE PRINT FOR READ), and 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 MANGANESE 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.")

Extensions and modifications were provided for in a variety of ways.

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/expressions. For example, a computed edit macro produces a list of edit commands, and a computed file package command 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 their 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.[11] 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.[12] AltoLisp introduced the idea of providing a microcoded target language for Lisp compilations, which modeled the basic operations of Lisp more closely than could a general-purpose instruction set.[13] AltoLisp served as a model and departure point for Interlisp-D,[14,15] the implementation of Interlisp for the Dolphin and Dorado Xerox personal computers,[16] the successors to Alto. Interlisp-D now supports a 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 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."[7] 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.

Interlisp was not designed, it evolved—but this was the right approach.

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 the personal computing environment. Interlisp evolved in a timeshared, hard-copy terminal world, and vestiges of this heritage have carried over into implementations 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."[7]

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.

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, a program that updates a Masterscope data base, or one that performs compilations. Personal computing thus causes a qualitative change in the programming environment, because the machine can be working continually for a single user.

Integration of the display. A significant addition to Interlisp on the new generation of personal computers, such as Interlisp-D,[14] 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 is an experimental system that explores some of these techniques.[17] 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 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

1. E. Charniak, C. Riesbeck, and D. McDermott, *Artificial Intelligence Programming,* Lawrence Erlbaum Associates, Hillsdale, N.J., 1979.
2. D. Friedman, *The Little LISPer,* SRA Pub., Menlo Park, Calif., 1974.
3. E. Shortliffe, *Computer-Based Medical Consultations,* American Elsevier, New York, 1976.
4. R. S. Boyer and J. S. Moore, *A Computational Logic,* Academic Press, New York, 1979.
5. J. Wolf and W. A. Woods, "The HWIM Speech Understanding System," in *Trends in Speech Recognition,* Prentice-Hall, Englewood Cliffs, N.J., 1980.
6. Warren Teitelman, "Toward a Programming Laboratory," *Proc. First Joint Conf. Artificial Intelligence,* May 1969, pp. 1-8.
7. Erik Sandewall, "Programming in an Interactive Environment: The LISP Experience," *ACM Computing Surveys,* Vol. 10, No. 1, Mar. 1978, pp. 33-71.
8. W. Teitelman et al., *The Interlisp Reference Manual,* Xerox Palo Alto Research Center, Palo Alto, Calif., revised Oct. 1978.
9. Warren Teitelman, "Do What I Mean," *Computers and Automation,* Apr. 1972.
10. Warren Teitelman, "Automated Programming—The Programmer's Assistant," *AFIPS Conf. Proc.,* FJCC, Dec. 1972, pp. 917-922.
11. D. G. Bobrow and D. L. Murphy, "The Structure of a LISP System Using Two Level Storage," *Comm. ACM,* Vol. 10, No. 3, Mar. 1967, pp. 155-159.
12. C. P. Thacker et al., *Alto, a Personal Computer,* Xerox Palo Alto Research Center, Report No. CSL 79-11, Palo Alto, Calif.,1979.
13. L. P. Deutsch, "A Lisp Machine with Very Compact Programs," *Proc. 3rd Int'l Joint Conf. Artificial Intelligence,* Stanford University, Palo Alto, Calif., 1973.
14. R. R. Burton et al., "INTERLISP-D: Overview and Status," in *Papers on Interlisp-D,* Xerox Palo Alto Research Center, Report SSL-80-4, Palo Alto, Calif., Sept. 1980, pp. 1-10..

15. R. R. Burton, "INTERLISP-D Display Facilities," in *Papers on Interlisp-D,* Xerox Palo Alto Research Center, Report SSL-80-4, Palo Alto, Calif., Sept. 1980, pp. 33-46.

16. B. W. Lampson and K. A. Pier, "A Processor for a High-Performance Personal Computer," *7th Int'l Symp. Computer Architecture,* La Baule, France, May 1980.

17. Warren Teitelman, "A Display Oriented Programmer's Assistant," *Proc. 5th Int'l Joint Conf. Artificial Intelligence,* 1977, Vol. II, pp. 905-915.

Warren Teitelman is a researcher at Xerox Palo Alto Research Center, where he was instrumental in the development of Interlisp. He also designed and implemented DLisp, an extension of Interlisp that makes extensive use of a bit-mapped display and a pointing device. Prior to joining Xerox, Teitelman worked at Bolt Beranek and Newman, where he coordinated the development of BBN Lisp and was responsible for its documentation and most of its interactive features. Work in these areas is a natural outgrowth of his longstanding interest in making programming systems easier to use and more accommodating to the demands of users.

He received degrees in mathematics from the California Institute of Technology (BS, 1962) and from the Massachussetts Institute of Technology (MS and PhD, 1966 and 1972, respectively).

Larry Masinter is a member of the research staff at Xerox Palo Alto Research Center. His research interests include interactive programming tools, the programming environment, and their implementations on new computers.

Masinter, a member of ACM, completed a BA in mathematics at Rice University in 1970 and a PhD in computer science at Stanford University in 1980.