Palo Alto Research Center

The Cedar Programming Environment: A Midterm Report and Examination

Warren Teitelman

XEROX

The Cedar Programming Environment: A Midterm Report and Examination

Warren Teitelman†

CSL-83-11 June 1984 [P83-00012]

CR Categories and Subject Descriptors: D.2.6 **[Software Engineering]:** Programming environments.

Additional Keywords and Phrases: integrated programming environment, experimental programming, display oriented user interface, strongly typed programming language environment, personal computing.

† **The author's present address is:** Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, Ca. 94043. The work described here was performed while employed by Xerox Corporation.

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

Abstract:

This collection of papers comprises a report on Cedar, a state-of-the-art programming system. Cedar combines in a single integrated environment: high-quality graphics, a sophisticated editor and document preparation facility, and a variety of tools for the programmer to use in the construction and debugging of his programs. The Cedar Programming Language is a strongly-typed, compiler-oriented language of the Pascal family. What is especially interesting about the Cedar project is that it is one of the few examples where an interactive, experimental programming environment has been built for this kind of language. In the past, such environments have been confined to dynamically typed languages like Lisp and Smalltalk.

The first paper, "The Roots of Cedar," describes the conditions in 1978 in the Xerox Palo Alto Research Center's Computer Science Laboratory that led us to embark on the Cedar project and helped to define its objectives and goals. Important decisions had to be made about what facilities and features were essential versus simply desirable, both with regard to the programming language as well as tools and packages. This section not only presents these decisions, but also describes the process by which we reached them. These deliberations are especially interesting in light of the fact that three communities with diverse programming languages (Mesa, Lisp, and Smalltalk) and very different programming *styles*, met to discuss the merits and drawbacks of their individual systems and religions, with the purpose of reaching some sort of consensus that would allow the construction of a programming environment that would be satisfactory to all of them.

The second paper, "A Tour Through Cedar," is essentially a travelogue through the current Cedar environment (as of September, 1983) in the form of a transcript of an actual session. This transcript consists of numerous snapshots of the display screen interspersed with dialogue and commentary. The intent is to produce an effect similar to that of the reader sitting down with a user in front of a display terminal and being given a live demonstration of the system, while an expert comments on some of the why's and wherefore's. During the course of this demonstration, the reader is introduced to most of the salient features of the Cedar Programming Environment as they come up and are used. In many cases we will digress from this demonstration to discuss some aspect of these features, such as why we did it this way, how important this particular facility actually turned out to be, etc.

The final paper, "Cedar: The Report Card," discusses and attempts to evaluate how well we have succeeded in reaching our objectives and goals, to what extent the original objectives and goals were changed or evolved during the course of the project, and what remains to be done.

A version of the paper "A Tour Through Cedar" appeared in *IEEE Software*, April 1984.

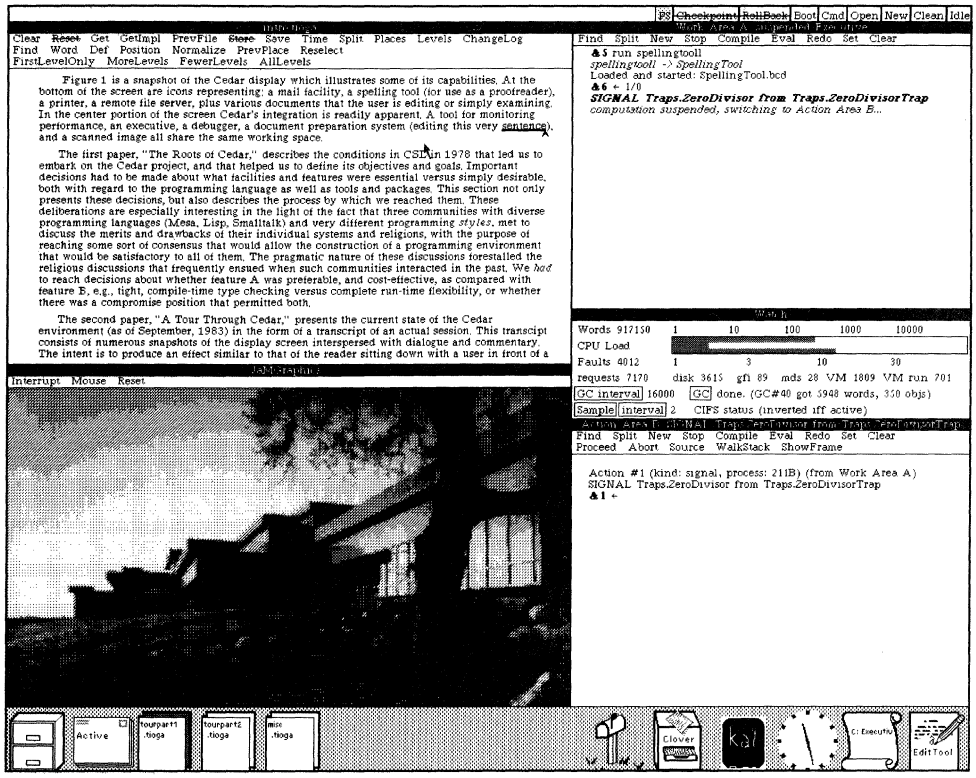
Introduction

A major activity in the Computer Science Laboratory (CSL) of the Xerox Palo Alto Research Center (PARC) is the production of prototype systems that provide interactive, personal computing services. Some of these systems are short-lived experiments to test novel ideas, and some are solid systems that are used by many people. The ability to conduct such experiments quickly, and at low cost, is thus of paramount importance to CSL: "The software that we can produce, and the rate at which we can produce it, are too often limiting factors in our research. ... We believe that it is increasingly desirable, feasible, and economic to use computers to directly assist the process of *experimental programming.* [by which we mean] ... the production of moderate-sized systems that are usable by moderate numbers of people in order to test ideas about such systems. We believe that it will be important to conduct future experiments more quickly and at lower cost than is possible at present" [8].

This belief provided much of the initial impetus for Cedar, a major project now under way in CSL to develop an advanced, integrated programming environment. Cedar is a programming environment designed to help programmers build experimental systems. It is the software equivalent of the kind of machine shop needed by an engineering laboratory, but unlike a machine shop, it does not represent a known technology: Cedar is itself an experimental system, and a very large one.

The main goal of Cedar is to increase programmer productivity, specifically the productivity of the programmers in CSL, by reducing the *cost* of solving a problem by software and by improving the *quality* of the solutions. The improvement will come from three main sources: a programming language that takes more responsibility for certain programming tasks, programming tools that make program development and debugging faster, and a package library that allows programmers to build upon one another's work. [21] We intend for Cedar to be the basis for most of our programming during the next several years. Cedar will also provide the platform for these experimental applications themselves, i.e., the applications that we develop will not only be constructed using Cedar, they will run on top of Cedar. Cedar will also support non-programmers, and programmers when they are not programming, by providing various office-related facilities such as an electronic mail system, a sophisticated editor and document preparation system, and a high-quality typesetter.

This report consists of three papers on Cedar, a system which is now in its adolescence: while there are approximately thirty serious users of the system, most of whom are extremely satisfied and view the environment as a vast improvement over the way they used to operate, a number of the facilities that we envision have not yet been provided, and others need significant improvements. Nevertheless, we feel that Cedar is a state-of-the-art programming system. It combines in a single integrated environment: high-quality graphics, a sophisticated editor and document preparation facility, and a variety of tools for the programmer to use in the construction and debugging of his programs. The Cedar Programming Language is a strongly-typed, compiler-oriented language of the Pascal family. What is especially interesting about the Cedar project is that it is one of the few examples where an interactive, experimental programming environment has been built for this kind of language. In the past, such environments have been confined to dynamically typed languages like Lisp and Smalltalk. (Not surprisingly, we have drawn heavily on our experiences with these latter two environments in the design of Cedar.)



The Cedar Display

The above snapshot of the Cedar display illustrates some of Cedar's capabilities. At the bottom of the screen are icons representing: a mail facility, a spelling tool (for use as a proofreader), a printer, a remote file server, plus various documents that the user is editing or simply examining. In the center portion of the screen Cedar's integration is readily apparent. A tool for monitoring performance, an executive, a debugger, a document preparation system (editing this very sentence), and a scanned image all share the same working space.

The first paper, "The Roots of Cedar," describes the conditions in CSL in 1978 that led us to embark on the Cedar project, and that helped us to define its objectives and goals. Important decisions had to be made about what facilities and features were essential versus simply desirable, both with regard to the programming language as well as tools and packages. This section not only presents these decisions, but also describes the process by which we reached them. These deliberations are especially interesting in the light of the fact that three communities with diverse programming languages (Mesa, Lisp, and Smalltalk) and very different programming *styles*, met to discuss the merits and drawbacks of their individual systems and religions, with the purpose of reaching some sort of consensus that would allow the construction of a programming environment that would be satisfactory to all of them. The pragmatic nature of these discussions forestalled the religious discussions that frequently ensued when such communities interacted in the past. We *had* to reach decisions about whether feature A was preferable, and cost-effective, as compared with feature B, e.g., tight, compile-time type checking versus complete run-time flexibility, or whether there was a compromise position that permitted both.

The second paper, "A Tour Through Cedar," presents the current state of the Cedar environment (as of September, 1983) in the form of a transcript of an actual session. This transcript consists of numerous snapshots of the display screen interspersed with dialogue and commentary. The intent is to produce an effect similar to that of the reader sitting down with a user in front of a display terminal and

being given a live demonstration of the system, while an expert comments on some of the why's and wherefore's. During the course of this demonstration, the reader will be introduced to most of the salient features of the Cedar Programming Environment as they come up and are used. In many cases we will digress from the demonstration to discuss some aspect of these features, such as why we did it this way, how important this particular facility actually turned out to be, etc. A somewhat condensed version of this paper appears in the Proceedings of the Seventh International Conference on Software Engineering, March, 1984, Orlando, Florida, as well as in the April 1984 issue of IEEE Software.

The final paper, "Cedar: The Report Card," discusses and attempts to evaluate how well we have succeeded in reaching our objectives and goals, to what extent the original objectives and goals were changed or evolved during the course of the project, and what remains to be done.

"Those who cannot remember the past are condemned to repeat it." -- George Santayana

The Roots of Cedar

In 1978, the computing community at PARC consisted of three distinct cultures: Mesa, Interlisp, and Smalltalk. Both the Smalltalk and Mesa communities programmed primarily on the Alto, a small personal computer that had been developed at PARC [32]. The Interlisp programmers continued to operate on a time-shared, main-frame computer called MAXC, a home grown machine that emulated a PDP-10 and ran Tenex. (An existence-proof implementation of Interlisp on the Alto had been completed, but performance problems, aggravated by the small size of the machine, were so severe that no one actually used this system to get serious work done.) Each of these communities was beginning to run into the limits imposed by the size of memory, both real and virtual, and by the computational power that the corresponding machine provided. CSL decided to solve these problems by designing and building a much more powerful personal computer, rather than by obtaining additional, more powerful time-shared, main-frame machines. To understand this decision, one must understand the unique (at that time) commitment at PARC to personal computing that began with the Alto computer.

Personal Computing at PARC

During early 1973, members of CSL, in consultation with others at PARC, designed the Alto computer system as an experiment in personal computing: "to study how a small, personal machine could be used to replace facilities provided only by much larger shared systems." Although the original design of the Alto was modified several times between 1973 and 1978 in order to increase its memory capacity and reduce its cost, the basic capabilities of the machine remained essentially the same.

The primary goal in the design of the Alto was radical (remember this was 1973): "to provide sufficient computing power, local storage, and input-output capability to satisfy the computational needs of a single user" [32]. The standard Alto system consisted of an 808 line bit map display, a keyboard and mouse pointing device, a 2.5MByte disk, an interface to the Ethernet, a microprogrammed processor that controlled input-output devices and allowed a number of instruction sets to be emulated, and 64K 16 bit words of memory.

As an experiment in personal computing, we considered the Alto a success.†1 The combination of high-resolution display and mouse pointing device provided a high-bandwidth, comfortable user interface. The Alto provided extremely reliable service as part of a distributed system: if one Alto broke, the user simply took out his disk pack and moved to another machine. The capabilities provided to the user by his personal Alto were supplemented by a large number of services available over the network, such as printing, electronic mail, and access to bulk file storage devices. This also led us into developing expertise in distributed computation, which was to become an integral part of our milieu.

The introduction of time-sharing in the early 1960's provided not only an economic way of utilizing the computing power of the large, expensive main-frames, but also produced a *qualitatively* different way of computing. Similarly, the introduction of the Alto significantly changed not only the way we used computers, but the way we thought about how computers should be used. For example, it was wryly observed that one of the best features of the Alto was that "it did not run faster at night." In other words, users of a time-shared machine encountering the tremendous competition for cycles during rush

†1 In fact, CSL almost suffered a success disaster. The Alto was so successful that other organizations within Xerox clamored for them, resulting in increased demands for CSL to provide support and software maintenance that threatened to interfere with our ability to do research. By the time the experiment was over, the number of Altos in use within Xerox had exceeded the original estimates of its designers by nearly two orders of magnitude!

hours often responded by working at odd hours of the night. These same users now found that they could get just as much work done on their personal machine during the day. Having everyone around at the same time is of considerable benefit to a research laboratory where chance interactions in the halls often lead to serendipity.

Another advantage of the Alto over a time-shared machine was that its personal nature made it socially acceptable for applications to devote the entire machine to interacting with a single user. This enabled a variety of real-time, interactive applications and tools to be built, such as illustrators and What-You-See-Is-What-You-Get text editors.

But there were aspects of the Alto design that did not work out well. In particular, the limitations on the size of the address space and on the amount of real memory were serious.†2 As a result, a great deal of time was spent squeezing software into the limited space available. However, we did not take this as an indictment of personal computing, but merely an indication that we had to think bigger.

The Dorado

By 1976, there were many hundreds of Altos in use within PARC and elsewhere. In CSL, Altos comprised the lion's share of our hardware base. But, it was evident that "a large and rapidly increasing amount of effort was going into surmounting the Alto's limited speed and storage capacity, rather than trying out research ideas in experimental systems" [15]. After much painful deliberation, we decided to design a new machine, the Dorado, to overcome these obstacles.†3 We intended that the Dorado would provide the hardware base for the next generation of computer system research at PARC.

Our requirements were that the Dorado had to rapidly execute programs compiled into a sequence of byte codes and also support high-bandwidth input/output. "In particular, color monitors, raster scanned printers, and high-speed communications [were] all part of the computer research activities; these devices typically have bandwidths of 20 to 400 million bits per second. [Such] fast devices must not excessively degrade program execution, even when the two functions compete for many of the same resources. Relatively slow devices, such as a keyboard or an Ethernet interface, must also be supported cheaply, without tying up the high-bandwidth i/o system" [7]. Furthermore, the Dorado had to be physically small and quiet enough so that it could sit in a user's office, and cheap enough that we could acquire them in significant numbers. (Cheap is a relative notion: the Dorado is still the most expensive personal computer ever built!)

These requirements resulted in a machine with a 60ns microcycle and an input/output bandwidth of 530 megabits/sec. (By comparison, the Alto had a 170ns microcycle and an input/output bandwidth of 32 megabytes.†4) Most Dorados currently have 2 megabytes of main storage expandable to 8 megabytes. (The Alto had 128K bytes of memory, later expanded to 512K via additional memory banks.) Dorado configurations range from our current 24-bit virtual address space (the addressable unit of the Dorado

---

†2 It is only fair to add, in the designers' defense, that these limitations became severe only after the Alto system had outlived its planned lifetime.

†3 The reason this decision was a difficult one was that designing and building the Dorado would require the full-time commitment of a number of our scientists for whom such activity would be a significant departure from their research interests. In effect, we were asking these individuals to sacrifice their opportunity to do interesting research for several years in order that the majority of the laboratory might benefit.

†4 Comparing the microcycle times of the Alto and Dorado may be misleading. The Alto was a lot less suited than the Dorado for interpreting the Mesa instruction set: the Dorado could execute a macroinstruction such as load or store in a single microcycle whereas on the Alto it might take a dozen microcycles. Furthermore, the Dorado had an instruction fetch unit that fetched instruction bytes from a stream and decoded them in parallel with, and independently of, the execution unit. The Dorado also had a hardware stack and hardware virtual memory. All of this resulted in an effective speedup of between a factor of ten and twenty over the Alto for strictly compute-bound programs (i.e., not accessing the disk).

is a 16-bit word) with two million bytes of main storage on up to the ultimate 28-bit virtual address with 32 million bytes of storage. A rough approximation is to think of a Dorado as having the power of an IBM 370/168 processor, dedicated to a single user. Although so powerful a machine could easily support several users and still give each much more computing power than had been provided by the Alto, we steadfastly resisted acting on this observation. The Dorado was designed and intended to be a powerful but *personal* computing system: as a workstation, it would support a single user.

In 1977, implementation of the laboratory prototype for the Dorado was begun. In the summer of 1978, this prototype successfully ran (by emulation) all of the Alto software. In the spring of 1979, Dorado Model 1 was pronounced operational. Currently, 75 Dorados have been built. In CSL, we have 25 personal Dorados and 6 pool dorados. By mid-1984, all CSL research scientists will have their own personal, dedicated Dorados.

Why Yet Another Programming Environment?

The arrival of the Dorado in 1978 resolved our immediate hardware problems; execution speed, memory size, and address space would not be issues for the foreseeable future. Thus, our ability to experiment with computer systems was now limited only by our programming capabilities, of which the principal component was the programming environment.

At this point in time, CSL had two major programming environments, Interlisp on MAXC [30] and Mesa on the Alto [20]. (Smalltalk [13] was also available, but its user community inside CSL was quite small.) Interlisp was a teletype-oriented environment, although an experimental window-oriented system had been designed and implemented [29]. Furthermore, the Interlisp model was of a single thread of control: there was no scheduler or support for concurrent operations.†5 Mesa, on the other hand, having evolved on the Alto, was display oriented. It also provided support for concurrent operations. But Mesa, had been severely stunted, much like a bonsai tree, by the nature of the Alto, particularly its hardware limitations. Thus, neither of these environments could really exploit the capabilities of the Dorado without a lot of work. At the same time, we were becoming increasingly concerned that having two programming environments was leading to duplication of effort and an inability to share results, especially programs. We saw the arrival of the Dorado as a golden opportunity to remove many of the limitations of our existing environments, while at the same time unifying our programming activities, by providing a single programming environment for the entire laboratory.

To explore this possibility, the first CSL Experimental Programming Environment (EPE) Working Group†6 was chartered and met intensively for a month during the summer of 1978. The results of this group are detailed in [11] and [8] from which much of the following material has been extracted.

The First EPE Working Group – Should We Do It?

Each of the members of the EPE Working Group had experience with only one of our environments: Interlisp. Mesa, or Smalltalk. Given our diverse backgrounds, we needed to decide on a common measure for evaluating programming environments. What makes one programming environment better than another, and under what circumstances? (For example, a feature that facilitates rapid prototyping may actually be detrimental to the development of robust, long-lived software.)

---

†5 The two issues of concurrency and use of the display are subtly related. It is hard for a user to monitor and control several parallel operations in an environment where he interacts with the system in a linear, teletype-oriented fashion. It is only with the flexibility provided by a display-oriented system that concurrency becomes an attractive and useful feature.

†6 Peter Deutsch, James Horning, Butler Lampson, James Morris, Edwin Satterthwaite, and Warren Teitelman, with the occasional participation of Alan Perlis.

We decided to restrict our attention on the foreseeable needs within CSL in the next few years, paying particular attention to what we called *experimental programmin.* by which was meant "the production of moderate-sized systems that are usable by moderate numbers of people in order to test ideas about such systems" [8]. Underlying all of our discussions was the belief that we had to be able to perform such experiments more quickly and with less manpower than was possible with our existing environments.

Having agreed to focus on experimental programming, we then decided to proceed by producing a catalogue of programming environment capabilities that are desirable for experimental programming, guided by our experience with the three previously mentioned environments – both their strengths and weaknesses. The catalogue would also include an estimate of the value, cost, and priority of each proposed capability. We would then use this catalogue as a basis for deciding whether it was feasible to build a single environment which included all or most of these capabilities.

Note that the existence of the Dorado meant that we were considering capabilities for a *computationally rich* environment: many of the capabilities on the list that we generated are feasible only when each programmer has substantial computing power available at all times.

Catalogue of Programming Environment Capabilities

To facilitate discussion and evaluation, we divided the capabilities of a programming environment into four categories: virtual machine/programming language, tools, packages, and other. *Virtual machine/programming language* was the most basic category and referred to those capabilities that were primitive concepts in the programming language, or in the virtual machine on which the programming language runs. Examples of such capabilities include object management (garbage collection), statically checked type system, abstraction mechanisms (explicit notion of interface),†7 adequate runtime efficiency, large virtual address space ($\geq$ 24 bits), run-time availability of all information derivable from source programs, good facilities for processes, monitors, interrupts,†8 and exception handling.†9 A total of 31 items were generated in this category.

The *Tools* category referred to capabilities employed by the user in dealing with his programs. Examples of tools are a prettyprinter, a source-language debugger, a cross-reference capability, and an available interpreter. *Packages* referred to programs that could be used by other programs. Examples of packages are a generalized cache mechanism, menus and other standard user interfaces, remote file storage, and a message transmission system. (The division between Tools and Packages was admittedly somewhat arbitrary since, in a good environment, the capability provided to the user by the tools would also be available to the programmer in the form of packages.) The *Other* category included documentation as well as non-technical considerations.

We then arrived at a priority ranking for these capabilities by giving each member of the working group 100 votes to be divided among the list. Each item was also rated as to how fundamental it was, in the sense of how difficult it would be to add that capability if it were not allowed for in the initial system design. We also estimated the difficulty of providing each capability in Mesa, Interlisp, and Smalltalk. (All of this information can be found in [8].)

---

†7 The importance and role of a statically checked type system and the explicit notion of an interface are discussed further in the subsection entitled "If We Start With Mesa" (page 14).

†8 Many members of the group felt that synchronization between logically asynchronous processes would be necessary for many of the applications we wanted to build, either for functional reasons or for efficiency, and that the programming language should provide mechanisms that would help the programmer to write such programs.

†9 Similarly, our experiences with Mesa suggested that an integrated mechanism for handling exceptional conditions aided the construction and debugging of robust programs by clearly distinguishing between *normal* and *exceptional* algorithms.

The complete catalogue of capabilities may be found in Appendix 1. It is interesting to note that there are very few surprises on this list: Despite our diversity, we were pretty much in agreement about what programming capabilities were desirable. The principal differences of opinion related to priorities (or difficulty of implementation). Appendix 2 contains the catalogue ranked according to priorities.

Fundamental Principles

In addition to the catalogue of specific capabilities shown in Appendix 1, in the course of our deliberations we also identified a few principles that were not captured as features, or else were so fundamental in our deliberations that they are worth repeating here:

> Automatic storage deallocation is an absolute necessity. It produces a qualitative improvement in the ease of programming and the reliability of the results.†10 This need not preclude another class of variables with programmer-managed deallocation, but the latter must not be able to destroy the data structures necessary for the former.
>
> Easy use of programs as data underlies many other facilities in the system. Implementing this seems to require having Lisp-style atoms, a run-time type system, and universal pointers (pointers that carry a type with them).
>
> Editing facilities closely coupled with the compiler and the executive interface to the system are essential, both to reduce the turnaround time for minor changes, and to allow easy construction of tools that interact with these facilities.
>
> Capabilities for precisely defining interfaces, and restricting the communication between modules to those interfaces, are essential to reliable and readable programming. This includes, as a special case, the ability to restrict the use of types and names to a local lexical context.
>
> The ability to perform static checking ... over designated program regions is necessary for both security and efficiency reasons.
>
> The present Lisp, Mesa, and Smalltalk programming styles *all* must be supported in a satisfactory way. The same packages, and tools of equivalent power, must be available in all styles. In particular, the Lisp capabilities for embedded variant languages and for programming entirely without type declarations must be supported.
>
> Both large, multi-person and small, single-person styles must be supported well. Bringing a wider range of experiments within the scope of a single person's effort is an important EPE goal.
>
> The EPE must support a wide range of binding times, including the Mesa and Smalltalk extremes [compile-time binding for Mesa and run-time binding for Smalltalk], in a way that allows changes in binding time without structural changes in the program. Different choices of binding time by the programmer may lead to different turnaround times for apparently minor changes, and to different execution efficiencies, but the *functional* behavior of programs must not depend on such choices. [8]

---

†10 It is not surprising that those members of the group with experience in the Lisp world would consider garbage collection non-negotiable. However, those from the Mesa community, which did not have garbage collection, were also becoming increasingly aware of its importance. Storage leaks, i.e., inappropriate retention of explicitly allocated storage, and the inverse problem of premature deallocation were the cause of some of the most pernicious bugs in Mesa programs, and some of the most difficult to find. Furthermore, many Mesa programs would become considerably simplified with the availability of automatic storage management and, for applications that employed parallel processing, a lot of synchronization issues would disappear.

Conclusions of the First Working Group

The EPE working group recommended that "CSL should launch a project on the scale of the Dorado†11 to develop a programming environment for the entire laboratory starting from either Lisp [Interlisp] *or* Mesa" [11].†12 We also concluded that the laboratory could support only one major programming environment. If a new programming environment were to be developed, "then other work on the existing systems must be kept to a minimum during this development, and support for these systems must be phased out as the new environment became viable" [11]. Finally, we observed that there did not seem to be any reason to attempt a multi-language programming environment. If we succeeded in constructing an environment with all or most of the capabilities in our catalogue, that environment would certainly support any existing style of programming. Furthermore, although the result might not satisfy everyone, an attempt to do so might cause the system to collapse under its own weight (cf. Lisp 2 and Algol 68).

The Second EPE Working Group – Where To Start?

From its creation in 1970, CSL had always been willing and able to take a long-term approach in planning and building for its future. We all felt that the laboratory as a unit, as well as most of the individuals that comprised it, were not merely transients; we were comfortable with investments in software and hardware that might take years to come to fruition. Mesa is a good example of the former, and the Alto and the Dorado examples of the latter. Thus, it is not surprising that the laboratory accepted the recommendation of the first EPE working group to develop a new programming environment for the Dorado starting from either Interlisp or Mesa. A second working group was formed†13 and met weekly for almost three months, exploring in more detail the "probable consequences of the alternative starting points." The results of this group are detailed in [11] from which much of the following material has been extracted.

The discussions of the second working group focused on four areas:

> What are the key technical issues that arise from each of the possible starting points?
>
> What differences in the ultimate system would necessarily follow from each choice of starting point?
>
> How much effort would be required to reach various levels of result from each starting point? What people would be available to do the work?
>
> What other non-technical considerations should play a significant role in our decision about what to do? [11]

---

†11 i.e., around 20 man years. We were a little optimistic; the current estimate of man years spent on the Cedar project, as of June 1983, is 45.

†12 Fairly early in our discussions we decided that either Interlisp or Mesa would be sufficiently preferable to Smalltalk as a starting point that we did not consider Smalltalk further. This decision was based on the following: (1) The Smalltalk 76 implementation was written in Alto assembly language and could not support more than 128K of real memory or 64K addressable objects without a major redesign and reimplementation. (Note that this was in 1978. Smalltalk 80, which was in fact a major redesign and reimplementation of Smalltalk, has removed these limitations [10].) (2) Many members of CSL were familiar with either Interlisp or Mesa, whereas there was no corresponding user community within CSL familiar with Smalltalk; and (3) the direction of Smalltalk evolution was towards smaller machines, whereas we wanted to take significant advantage of the Dorado's capabilities.

†13 Daniel Bobrow, Peter Deutsch, James Horning, Butler Lampson, Roy Levin, Larry Masinter, Gene McDaniel, James Mitchell, James Morris, Edwin Satterthwaite, Nori Suzuki, and Dan Swinehart. Many other members of CSL also attended some or most of the meetings and/or provided written material for the group.

Technical Issues

Most of our discussions centered around the first area: the technical issues that arose from the choice of starting with either Interlisp or Mesa. We addressed this question by taking the catalogue of programming environment capabilities and examining each of them in terms of the amount of effort required to provide the corresponding facility in Interlisp or Mesa. In many cases, the corresponding environment already provided that facility, so the effort would be minimal. For example, Interlisp already provided automatic storage management, and Mesa already had a statically checked type system. Conversely, adding automatic storage management to Mesa or adding encapsulation mechanisms to Interlisp was considered to be hard. Our discussions focused primarily on those key facilities missing from each environment. The next two sections examine each of the two environments in turn.

Mesa Facilities Needed in Lisp

The following capabilities from our catalogue were identified as being present in Mesa but not adequately available in Interlisp: statically checked type system, abstraction mechanisms (i.e., the explicit notion of interface), adequate run-time efficiency, encapsulation/protection mechanisms, consistent compilation, and user ability to pack data. A proposal was made to provide each of these facilities as follows:

> Static type checking -- by making types be objects and checking that all users of a declared variable refer to identical type objects.
>
> Explicit interfaces -- by defining an object called a dictionary that generalizes the notion of parameter list, record, and interface, and by requiring that such an object be associated with every defined or imported function.
>
> Greater run-time efficiency, packed data -- by making the internal representation of a quantity be one of its attributes in the type system, and extending the instruction set to allow more efficient execution when more tightly bound representations are used.
>
> Encapsulation/protection -- by associating protection information with entries in dictionaries.
>
> Consistent compilation -- by extending the notion of type identity and compatibility into the permanent filing system, through an object called a permanent pointer. [11]

In the Lisp tradition, the method proposed to provide these facilities was to define new objects, e.g., for representing types, interfaces, name scopes, and then to define functions for manipulating them. For example, the following plan was set forth for implementing the statically checked type system. Types would be objects, just like integers or lists. There would be three kinds of types: *interface*, which describe how objects behave in general, *descriptive*, which specify predicates that hold true of objects in particular situations, and *representation*, which describe the bit patterns used to represent objects inside the machine. Interface types would be built up from elementary types by (possibly recursive) applications of various operators such as: ANYOF[t1...tn: Type], which produces a type for objects that can be of any of the types t1...tn; FUNCTION[argT,resultT: Type], which is a type for functions that take arguments of type argT and return result of type resultT; READONLY, UNIQUE, etc. Interface types could also be defined as *structures*, where a structure was a type that had named components, each of which may have type declarations. For example, an interface type in the Mesa sense is simply a structure type with fields that are of various function types. Under this scheme, type checking, conformity, and coercion could all be dealt with straightforwardly. The remaining missing facilities were dealt with in a similar manner. [11]

Lisp Facilities Needed in Mesa

The group distinguished two different kinds of facilities available in Interlisp but missing from Mesa:

> *Foundations*: basic, low-level facilities needed to match the functionality of Lisp. ...
>
> *Features*: equivalents for the Lisp Masterscope,†14 file package, programmer's assistant, Helpsys, and similar features, in a more general setting, together with improvements to the [Mesa] debugger and editor, facilities for specifying the construction of complex systems, and other "environmental" capabilities. [11]

In general, the group felt that those capabilities from our catalogue that fell under the features category presented no serious technical problems, i.e., were straightforward albeit non-trivial. We did observe that the right set of facilities for the EPE we were considering might look substantially different from existing facilities, given that we had the opportunity to redesign the user interface to take advantage of the high-bandwidth display. In fact, some of the new experimental applications being developed in our existing environments were already demonstrating this, such as the Smalltalk browser, DLisp, and the Mesa Tools environment. Thus, these capabilities might need a great deal of additional design work once the EPE project was under way. But there was no compelling reason why their absence from Mesa and presence in Lisp mitigated strongly in favor of the latter.

With regard to foundations capabilities, more detailed consideration and planning would be required. We identified the following as being present in Interlisp but not available in Mesa: automatic storage deallocation, fast turnaround for minor program changes, run-time type system and self-typing data, runtime availability of all source program information, compiler/interpreter available with low overhead at run-time, and program-manipulable representation of programs.

The group proposed a plan for adding each missing facility. For example, to provide for automatically deallocated objects, the plan was to use the Deutsch-Bobrow scheme for incremental garbage collection [9]. This required "knowledge of the location and type of every pointer to an automatically deallocated object, but the techniques are well-known and straightforward" [11]. Similarly, the Mesa compiler already built symbol tables that included a mapping between source identifiers and internal, unique identifiers for an individual module. Implementing the Lisp atom capability would simply require a global map into which each module's map would be merged when it was loaded.

Character of the Result

As a result of these deliberations, the working group concluded that the advantages accrued by choosing Interlisp over Mesa or vice versa were insignificant. In other words, the choice between Interlisp and Mesa could not be made "solely on the basis of technical considerations" [11].

> To within the uncertainty of estimation, both starting points present challenges of comparable difficulty, and would lead to systems of comparable utility, with comparable investments, over comparable time spans. [11]

Regardless of which system was chosen as the starting point, the group reaffirmed the goal of supporting the programming styles of *both* communities:

---

†14 Masterscope is an interactive program for analyzing and cross-referencing user programs. It contains facilities for analyzing user functions to determine what other functions are called, how and where variables are bound, set, or referenced, as well as which functions use particular record declarations. It maintains a database of the results of the analyses it performs, which the user may interrogate via a simple command language. Masterscope is interfaced with both the editor and the file package so that when a datum is changed or loaded, Masterscope knows that it must be re-analyzed [17], [14].

> We are agreed that it is necessary and feasible for an EPE based on either system to support comfortably the programming styles currently associated with Lisp and Mesa. ... The final system would provide all the facilities that present Lisp and Mesa users value highly: from Lisp, tools similar to the existing array of tools in the current Interlisp, and the ability to support integrated sublanguages and to delay bindings; from Mesa, the provisions for modularization with explicit interfaces, and the amenability to static checking. However, the system would probably retain some of the "flavor" of its starting point. [11]

This last sentence deserves further elaboration. What the group was saying was that depending on the system chosen for the starting point, there would be differences in the "character of the result." In fact, it was "the value judgments placed on these differences [that] generated most of the heat in our discussion" [11].

What would these differences be? The following two sections contain some of what the group thought the flavor of a Lisp-based or Mesa-based EPE would be, and the strengths and weaknesses that would result from starting from each.†15

If We Start With Lisp

To understand the flavor of a Lisp-based programming environment, one has to look at how such systems developed, and how they are typically used.

> Lisp systems have been used for highly interactive programming for more than a decade. During that time, special properties of the Lisp language have enabled a certain style of interactive programming to develop, characterized by powerful interactive support for the programmer, nonstandard program structures, and nonstandard program development methods. ...
>
> Lisp is used almost entirely as a research tool. ... The average Lisp user writes a program as a programming experiment, i.e., in order to develop the understanding of some task, rather than in expectation of production use of the program. The act of developing the program, not the act of running it (even for test data), constitutes the experiment. As a consequence, the program is likely to be large and complex, to undergo drastic revisions while it is being developed, and to be thrown away before it has been "completed" by conventional programming standards since it will already have served its purpose before then. [25]

Beau Sheil [27] calls this style of use:

> *exploratory programming*, the conscious intertwining of system design and implementation. ... 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 the design and program develop together.

Current Lisp implementations (and especially Interlisp) evolved in response to the need for programming environments that facilitated the exploratory programming style of use. The following aspects of Interlisp are especially relevant for supporting this style:

> Incremental: Small changes require only a small amount of work (both mental effort and real time). This is true for both ordinary Lisp programs and programs written in embedded languages. (Current implementations, unfortunately, have relatively weak tools for discovering whether one's changes are consistent.)

---

†15 At times in the discussion, it may seem that we are comparing the Mesa *language* with the Interlisp *environment*. This is appropriate: most of the focus of energy and effort in Interlisp has been on its environment, whereas Mesa's great strength is its language. We expected that if we started with Mesa, most of our efforts would be in environmental-related areas, whereas if we started with Lisp, a lot more of our effort would have to be devoted to language-related issues.

> Open: The basic facilities of Lisp are open to change at a very low level; one can modify the operation of the entire system from a user program. There is no distinction between system and user code, variables, name spaces, etc. (This is also a weakness, in that it is not uncommon to find that parts of the system make assumptions about each other that casual modifications violate.)
>
> Integrated: The user can slip relatively gracefully from procedures written in an embedded language back to procedures written in Lisp itself, and vice versa. Since the parser and interpreter are packages, code (in a variety of languages) can be stored in data structures and executed when retrieved.
>
> Aware of user activities: Standard packages in Lisp [Interlisp] keep track of new objects added to the system by the user on-line, and changed objects, and help the user keep track of his complex environment. Because all transactions with system objects (such as editing, creating new variables, etc.) are handled through an active intermediary and a set of functional interfaces, it is easy to provide all the "hooks" for complex assistants like Masterscope.
>
> Abstraction-based: The user is completely freed from concern with basic questions like the representation of integers and symbols, and the management of storage. (Most Lisp implementations, including the present Interlisp, have a related weakness in that they provide few mechanisms for attaining better efficiency even when the user knows full generality is not needed.) [11]

We expected that an EPE based on Interlisp would support the exploratory programming paradigm, i.e., facilitate rapid changes, better than one based on Mesa.

If We Start With Mesa

Mesa evolved in response to an entirely different need: producing reliable, robust systems, developed by large teams of programmers,†16 and the ability to maintain such systems over a fairly long period, often by programmers who were not the original implementors. The second EPE report described the Mesa style as follows [11]:

> The "Mesa style" tends to place greater emphasis on structure than on unconstrained flexibility. Probably its two most important aspects for an EPE are its emphasis on static checkability and its provision for explicit interfaces. Both are important in speeding up the programming process and in improving the quality of the result; they become even more so if the units that programmers manipulate are large (packages or subsystems), rather than small (statements or functions). The advantages will be small for programs whose "characteristic times" (design, programming, checkout, existence, total execution) are all measured in minutes, large if they are measured in weeks or months. In an environment where programs are undergoing rapid change, however, mandatory checking mechanisms tend to introduce unnecessary overhead by requiring complete internal consistency at every step of the development process.

Note that in sharp contrast to adherents of the Lisp style, proponents of Mesa are perfectly willing to accept the greater inertia to change imposed by the mandatory checking mechanisms. In fact, many would consider this to be a *feature* of Mesa, just as the fact that amending the Constitution of the United States is an extremely difficult and lengthy process is a feature because it tends to ensure that only well thought out and mutually agreed upon changes are implemented.

---

†16 It is unusual for a programming project in Lisp to involve more than three or four programmers, whereas there were as many as 35 programmers at work at one time on the Star project developed at Xerox.

Static Checkability. On the issue of static checking versus run-time checking, arguments about the advantages and disadvantages of both have raged back and forth for some time. The proponents of Mesa argue that it is better to locate faults by static checking than via run-time errors because [11]:

> Many faults can be identified in a single run of the checker, rather than surfacing one at a time in debugging runs.
>
> There is experimental -- as well as anecdotal -- evidence for the proposition that program faults located statically are diagnosed and removed more quickly than those located dynamically.
>
> Passing the static test ensures that *all* faults in a given class have been removed; in general, no finite set of test cases gives such assurance with dynamic checking.
>
> "Correctness" is a static property of the program text; it is hard to ensure that a program that relies heavily on dynamic properties actually does what is intended. ...
>
> The belief that this style actually speeds programming, measured as problem solutions per unit time, is closely tied to the observation that most programmers spend more time worrying about the possibility of programming errors -- and coping with their consequences -- than they spend actually writing new code. This style does require more planning before a running program is created -- some would consider this a disadvantage. A definite weakness of this approach is that it will be relatively difficult to add flexibility that was not anticipated in the program design, thus restricting the range of experiments that can be easily performed.

The importance of detecting faults earlier rather than later is captured in an aphorism attributed to James Morris: "There is no debugger in Peoria." If a program is to be run by unsophisticated users at sites distant from its implementors, encountering problems at that time is simply not acceptable. On the other hand, most Lisp programs are written as experiments: they may never be run in Peoria (though this is beginning to change). Thus, it is of no consequence that there may be bugs lurking in as yet untried portions of such a Lisp program; the Lisp programmer is happy to defer dealing with those problems until he encounters them. In fact, it is a *feature* of Lisp that programs can be run when they are only partially complete.

Another aspect of the issue of when it is best to detect errors is that because it is relatively hard to change Mesa programs (edit, recompile, reload, etc.), it is important to detect as many faults as possible in a program before it is run, and to fix them all at once. In other words, precisely because the consequence of encountering a problem is more serious in the Mesa environment than in Lisp, it is more important to find at one time as many problems as possible. By contrast, when a problem is encountered in a Lisp program, e.g., a type conflict, it is usually a simple matter to fix it immediately and even to continue with the computation. (Sometimes the programmer may have to backtrack the computation to get to a consistent state, but most Lisp debuggers provide such facilities.) Thus, encountering problems as a sequence of isolated, individual events during the course of testing a program is perfectly acceptable to the Lisp programmer.

Explicit Interfaces. Much of the original motivation behind the development of Mesa was to facilitate a modular style of programming. Thus, a great deal of thought has been placed on the issue of explicit interfaces in its design. The separation of Mesa programs into interfaces and implementations of these interfaces enable implementors and clients to work independently, and to make changes independently, as long as they respect the interface. As stated in the first EPE report [8]:

> Abstraction mechanisms are important because they make explicit the logical dependencies of one part of a program on another, while concealing implementation choices irrelevant to the communication between such parts. Thus, these mechanisms enable the system architect to factor the development, debugging, testing, documentation, understanding, and maintenance of programs into manageable pieces, while leaving individual programmers the appropriate freedom to design those pieces.

Of course, (just as with static checkability) the need to specify interfaces in advance can be viewed as either a strength or weakness of Mesa, depending on one's point of view.†17

Early versus late binding. The combination of static checkability and explicit interfaces encourages, indeed, often requires, relatively early binding of many aspects of Mesa programs that in Lisp are typically bound during execution. This is in sharp contrast to the approach taken by Lisp and other programming languages designed for exploration:

> 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 interest of minimizing and deferring the cost of making large-scale program changes. ... [These] languages make extensive use of *late binding*, i.e., allowing the programmer to defer commitments as long as possible. [27]

Note that the Lisp programmer is not necessarily adverse to the availability of explicit interfaces or static checkability. He just wants them to be optional.†18

The bottom line is that we expected that an EPE based on Mesa would emphasize robustness, reliability, and maintainability at the expense of the ability to make rapid changes.

The Decision to Start With Mesa

The principal conclusion of the second working group was that the choice between Interlisp and Mesa as the starting point for Cedar could not be made solely on the basis of technical considerations. Thus, social and political factors, including the availability of people to fill key roles in the project, as well as our concern for relations with both the computer science research community and the rest of the Xerox Corporation, became decisive concerns. These considerations could be divided into the categories of importation and exportation – of ideas, people, and code. In many cases, symmetry prevailed: there were roughly the same kinds of advantages and disadvantages for Interlisp as Mesa. However, where there was an edge, it tended to go to Mesa. For example:

• The rest of Xerox had a fairly large and growing commitment to Mesa, and none to Interlisp. Remaining largely compatible with the rest of the corporation had both advantages and disadvantages, but the advantages predominated. With respect to research communities outside of Xerox, either choice would reduce communication with important (but different) research communities in the outside world: Lisp favors the AI community, Mesa favors the programming language and systems programming community.

• Although the efforts required were about the same, a somewhat larger number of qualified people were available to work on a Mesa-based EPE. It was noted, however, that if Mesa were chosen, some effort would be needed to ensure that those members of the Lisp community concerned with programmer assistance, programs as data bases, and integrated sublanguages were able to provide enough input to ensure that Cedar would be of use and attractive to them.

---

†17 The present need for recompilations of a large number of files whenever a fundamental interface is changed, even in an upwards compatible fashion, has to count as a weakness. But, it is simply an artifact of the current implementation, can certainly be reduced, and probably eliminated..

†18 For example, the Decl package developed by Ron Kaplan and Beau Sheil extends Interlisp to allow the user to declare the types of variables and expressions. "It provides a convenient way of constraining the behavior of programs when the generality and flexibility of ordinary Interlisp is either unnecessary, confusing or inefficient" [14].

• We wanted to keep abreast of developments in computer science in the world at large. Most of the knowledge representation, expert systems, automatic programming, and programmer's assistant type of work is done in various Lisp dialects. However, much of the formal specification and verification work was directed towards Pascal dialects, and would therefore be more easily applicable to Mesa. Also Ada being based on Pascal, is much more similar to Mesa than to Lisp, so that work on Ada environments would be highly relevant.†19

• We wanted to move implementors into the project easily, and were even more concerned that it be easy for users to convert to the *use* of Cedar.†20 Within CSL, there were roughly comparable numbers of hard core users in each camp, so that the issues of migration seemed roughly symmetric. However, within Xerox, Mesa was much more widely known and used. Outside of Xerox, generic Lisp was more widely known than Mesa, but generic Pascal was more widely known and used than Interlisp.

As a result, the decision was made in early 1979 to launch a project to build Cedar, an experimental programming environment, starting with Mesa.

First Steps

1979. Work began on Cedar in 1979 with the creation of seven committees to investigate and propose implementation strategies for various components of the system: Language Features, User Facilities, Communications, Data Base Package, Text and Figure Displayer, Filing and Consistent Compilation, and Programmable Scanner Capabilities. However, with so large a system and so many diverse components (many of which involved research problems themselves), it was important that we did not simply disperse into independent efforts and then try to fit everything together at the last minute. We needed to define various points along the way when we would try to put together a working system in order to provide us with feedback, and also (hopefully) provide us with interim environments that would allow us at each stage to be more productive than before. Dependency was another issue; some components could not effectively proceed until others were in place or at least designed. Thus, the order in which we attacked various issues would be extremely important.

At this point in time, there was no integrated Mesa programming environment. Users composed and edited Mesa programs using a separate, stand-alone editor (Bravo), compiled them using the stand-alone Mesa compiler, and then loaded and tested them using the Mesa debugger. When problems were encountered, the user had to exit from the debugger, return to the editor, load the source files, make the changes, leave the editor, recompile the programs, and then reload and resume testing.

We decided that there would be sufficient short-term payoff in our productivity while building Cedar to spend some initial effort at bootstrapping ourselves into a better environment. Therefore, we constructed the Interim Mesa Environment (IME) specifically as a tool for building Cedar. IME's aim was to reduce the delays experienced in developing Mesa programs by providing support for the edit, compile, and load phases in a single environment. By the end of 1979, IME was in place and was being used seriously by about six people.

---

†19 Five years later, it now appears that the converse of this statement is going to be of greater importance, i.e., the fact that Cedar is built on top of Mesa makes Cedar highly relevant to Ada environments.

†20 CSL programmers have a wide range of goals and styles: system programming, mathematics, hardware design, artificial intelligence, etc. Most of them are permanent employees who can be expected to invest effort in learning how to use a programming system, but an appreciable fraction are visiting scientists or computer science students who can be productive only if they can quickly assimilate the basic system.

Meanwhile, we worked on specifying a subset of the Mesa language in which it was possible to do automatic storage deallocation. By the end of 1979, the design was complete and implementation begun. In the area of user facilities, we held substantial design discussions about the Cedar document metaphor (DOCs), which was intended to provide a standard mechanism by which data structures were displayed and changed. These discussions led to a detailed implementation design and we started the actual implementation of a number of document types.†21 We also designed, implemented, and tested a collection of graphics-oriented algorithms that would form the basis for the Cedar Display facilities. These included geometric transformation routines that implement arbitrary scaling, rotation, and translation, and clipping algorithms for text, lines, and curves.

1980. Up to this point, we had been using the Alto operating system, and emulating the Alto instruction set on our Dorados. Not only did the emulation cause performance inefficiencies, but the Alto operating system contained no support for virtual memory, and its file system was inadequate for our purposes (it only enabled addressing of about one fifth of the Dorado's 80 megabyte disk). We were faced with the choice of either implementing our own operating system or adapting the existing Pilot operating system [24]. We chose the latter course because of our limited manpower resources. (We revisited this decision in 1983.) By the end of 1980, we had modified the Pilot operating system to enable it to run on the Dorado, had produced an initial version of the Cedar kernel built on top of Pilot, and had succeeded in transferring all of our code to run on top of this kernel. We replaced IME by a system called Cascade, which allowed the user to edit, compile, bind, execute and debug Mesa programs under Pilot, i.e. without resorting to Alto emulation.

In the area of language development, 1980 saw many of the changes required for Cedar Mesa incorporated into the Mesa compiler and runtime system, including: reference-counted assignments, type equivalence, run-time type checking, and LISP-like atoms and lists. The incremental garbage collector, which would be the most critical component of the Cedar Mesa runtime system, was thoroughly tested, and its performance improved via special microcode. A trace-and-sweep garbage collector had been designed and an initial implementation tested.

In 1980, we also began work on the Cedar data base facility and system modeling. Much of the data base storage system and the lower levels of the system were implemented and extensively tested. A design document specifying the system modeling language was written. An interim facility was developed to create systems from their configuration files by performing all of the necessary file transfers, compilations, and bindings automatically. Besides its immediate usefulness, this facility contained many building blocks for the eventual system model implementation.

The design for a document preparation system called Tioga progressed significantly during 1980. Tioga would consist of a coordinated editor and typesetter to aid users in creating documents with high-quality typography. An implementation of the Tioga editor was begun.

We also investigated how the Cedar debugger, compiler, and run-time support would interact to enable evaluation of expressions at run time (i.e., an interpreter), to provide a powerful breakpointing facility, and to access/display values using the run-time type system. This investigation resulted in a set of interfaces for these tasks. We also began implementation of the base-level software for providing input events from the user (keystrokes, mouse positions and selections) to higher level software.

---

†21 DOCs proved to be too ambitious an undertaking for the time scale and resources available, and was subsequently abandoned in favor of the Viewers Window Package.

First Usable Systems Appear

1981. In January, we held a week-long review of the Cedar project by a panel drawn from various groups in the Xerox Palo Alto community. The reviewers' basic conclusions were that a significant amount of good work had been done on the many building blocks and it was time to pull a system together that could be used by programmers. Partly as a result of this recommendation, we spent significant effort during 1981 on integrating the useful pieces of Cedar, so that by mid-1981, approximately ten programmers were using the core facilities and five of them had begun to use the tools. By the end of the year, approximately 20 programmers were using Cedar, several of them for projects other than Cedar itself.

During 1981, we improved the performance and robustness of the allocator, garbage collector, and basic run-time type system. Critical parts of the allocator and run-time type discrimination machinery were microcoded on the Dorado, completing the planned Dorado microcode support for Cedar. We also developed software tools for measurement, and used them to make changes to improve system performance, including refinements to strategies for balancing allocation and collection activity, and to improve that part of the run-time that deals with the virtual memory allocator.

In the language area, the typed-value part of the Cedar abstract machine interface (for performing operations at run-time on the types of values) began to be used extensively. A facility for inserting break points into user programs was designed and implemented. We also identified a subset of the Cedar language, the Safe subset, in which incorrect programs could not interfere with the garbage collector, and implemented compiler-enforced restrictions for guaranteeing that programs remained in this subset.

In 1981, the Cedar Interim File System (CIFS) was designed and coded. A client of CIFS used a single mechanism to access the diverse set of storage facilities at PARC. The implementation of Tioga progressed substantially; we decided to use it as the program editor for Cedar. We began the design and implementation of Viewers, a high-performance, general-purpose window package for uniform screen management. BugBane, the Cedar debugger, was released for alpha testing. BugBane provided facilities for expression evaluation, stack display, uncaught signal handling, and breakpoints. Since it ran in the same address space as the program being debugged, user interaction was significantly faster than with previous debuggers.†[22]

The first version of the system modeler was completed and testing begun. Description Files (DF files), an interim file management tool for describing system components, was implemented and used for maintaining virtually all Cedar programs. By mid-1981, Cedar had grown sufficiently large and complicated that we implemented the Release Tool, a facility for automating the process of releasing Cedar. Based on an extension to DF files that permitted automatic consistency and completeness verification of an entire system, the Release Tool validated a candidate release and stored it in a safe repository. It made it possible for new releases of Cedar to occur at a rate of about once a month and greatly increased the dependability of the system. Cedar 2.0, the first release of Cedar using the Release Tool, took place in October 1981, and consisted of 22 components and approximately 1800 files.

---

†22 Up until this point, debugging in Mesa had been performed via a world-swap debugger. World-swap debugging is a debugging system that works by writing the real memory of the target system (the one being debugged) onto a secondary storage device, and reading in the debugging system in its place. The debugger then provides its user with complete access to the target world, mapping each target memory address to the proper place on secondary storage. This somewhat clumsy style of operating allows very low levels of a system to be debugged conveniently, since the debugger does not depend on the correct function of anything in the target, except for the very simple world-swap mechanism. However, it requires writing and reading all of real memory (2 megabytes) each time control is transferred from the client to the debugger, or vice versa. Thus, the availability of Bugbane meant that a simple operation like hitting a breakpoint and proceeding was reduced from on the order of ten seconds to on the order of one or two seconds.

The First Real System

By the end of 1981, we had set for ourselves the immediate goal of creating an environment that would be preferred to the Mesa/Cascade system (Tajo). This configuration, Cedar 3.0, would not initially dominate the Tajo environment in all respects; but the temporary lack of certain features would be overcome by other aspects of Cedar: garbage collection, run-time types, graphics facilities, system modelling, and fast turnaround for small program changes. We decided that other Cedar goals would be deferred until this goal was met.

1982: 1982 saw furious activity and growth in the Cedar project. Cedar 2.2 containing BugBane, the Cedar debugger, was released in January. Cedar 2.2 consisted of 26 components and was followed quickly by Cedar 2.4 (40 components) in February, which was the first Cedar release to include Viewers and Tioga. This marked the first release in which the hardy user could edit, compile, load, and run programs entirely within Cedar. In March, we demonstrated Cedar to approximately 150 attendees of the Symposium on Architectural Support for Programming Languages and Operating Systems. Their extremely favorable reaction provided us with a much needed morale boost. Cedar was becoming real.

Cedar 2.5 (48 components) was released in March and included the UserExec, an Interlisp-style executive complete with history and spelling correction. Finally, in May, Cedar 3.0 was released marking our official break with Tajo, which was eliminated entirely from the release. Cedar 3.0 involved 62 components, over 300,000 lines of source code, and over 4000 files (including both source and object files). By the end of 1982 when Cedar 3.5 was released, Cedar had grown to 96 components, 4800 files, 450,000 lines of code, and 300 pages of documentation.

During 1982, the Safe Subset of the Cedar language began to come into general use, thereby making programs less subject to hard-to-diagnose errors. The trace-and-sweep garbage collector was implemented; it was designed to recover unreferenced circular data structures and to serve as a reliable backstop for the incremental collector. The use of collectible storage was pushed lower in the system to support a new loader based on interface records. The abstract machine was extended to deal with breakpoints, processes, paving the way for the use of Cedar for both world-swap debugging and tele-debugging.†[23] During 1982, the Safe Subset of the Cedar language began to come into general use, thereby making programs less subject to hard-to-diagnose errors.

Meanwhile, we made significant improvements to BugBane, thereby allowing many users to avoid the use of world-swap debugging altogether for diagnosis of simple bugs. By the end of 1982, most Cedar users were using BugBane entirely for their debugging: world-swap debugging was used primarily by developers of the lower levels of the system. The interfaces between Bugbane and the UserExecutive were overhauled and the two facilities unified, yielding major improvements in the smoothness and robustness of the system. Bugbane was simplified to use the abstract machine implementation exclusively for access to symbols and system data structures. The UserExecutive was extended to incorporate a separate *action area* for each pending *action* (breakpoint, uncaught signal, etc.). All symbol and source files associated with a release system were made accessible to the debugger, whether or not copies resided on the local disk.

Version 1.0 of Tioga was released, marking the completion of a reliable, high-quality editor and page formatter that was now in general use. The Viewers Window Package was largely completed and also in general use. We had reached the stage where we had the luxury of beginning work implementing various experimental creature comforts, such as automatically generating and updating change logs when the user edits a file.

---

†23 Tele-debugging is a slight variation to world-swap debugging in which the debugger runs on a different machine with a small nub in the target world which can interpret *ReadWord*, *WriteWord*, *Stop*, and *Go* commands arriving from the debugger over a network.

Cedar 3.0 marked the first time that users could send electronic mail without leaving their Cedar environment. Cedar 3.3 contained the full Walnut mail system which allowed users both to send and receive mail, sorting their incoming messages into message sets stored in a data base which could be interrogated in various ways.

By the end of 1982, the catalog of Cedar packages included: graphics, data bases, performance measurement, screen management, lists, priority queues, symbol tables, remote procedure calls, random numbers, and file comparison. Documentation was under way. Applications were beginning to appear. Cedar was rapidly becoming an environment.

Today

1983. In March, we released Cedar 4.0, the first release in which Cedar facilities were used for both client and world-swap debugging. Having reached this plateau, we decided to embark on a project to design and implement the Nucleus, a replacement for the remaining major piece of software in Cedar which had not been implemented as part of the Cedar project itself, namely the Pilot operating system.†24 At the time of this writing, that project is nearing completion, and the next major release of Cedar, Cedar 5.0, will be built upon the Nucleus.

Meanwhile

Lest the reader get the mistaken impression that Cedar was the only programming environment (experimental or otherwise) being developed at PARC from 1978 to 1983, while CSL was engaged in developing Cedar, work on both the Interlisp and Smalltalk environments continued in other laboratories within PARC.†25 This was fortunate, for as discussed in "Cedar: The Report Card," the third paper in this report, Cedar did not achieve its initial goal of providing support for the Lisp and Smalltalk programming styles, and therefore never became an attractive alternative to these communities.

Smalltalk

In 1978, Smalltalk had finally completed the transition from the proof-of-concept provided by the Smalltalk-72 system to a mature programming environment, Smalltalk-76 [13]. Smalltalk-76 was the first system to introduce to the PARC community the notion of windows and menus. It featured a mouse-driven code browser capable of accessing the source code of all the procedures in the system, either locally or via Ethernet access to remote servers. Code and any other text could be easily modified by a point-and-type modeless text editor. Smalltalk-76 included a debugger that was fully integrated with the window system, gave immediate access to source code and variable values, and allowed in-place editing of code and resumed processing from the point of suspension. Owing to the modularity of Smalltalk's message-passing model (callers do not depend on callee), the total turnaround time for making a change to the system was under five seconds. Note that most of these features of Smalltalk were also goals of Cedar.

---

†24 There were two major reasons why it was important to eliminate Cedar's dependence on Pilot, both stemming from the fact that Pilot had as its primary purpose and orientation the support of a product. First, the maintainers of Pilot could not afford to be as responsive to our needs as we would like, since their software was tied to a product release cycle. Second, since the typical Cedar work station was so different from product work stations, both in terms of hardware, software, and style of use, many engineering decisions and tradeoffs appropriate for one application were simply wrong for the other. A good example is the demands placed on an operating system by the presence of garbage collection, a key component of Cedar.

†25 Although further work on the Mesa environment in CSL stopped, another organization within Xerox developed an integrated, interactive programming development environment for Mesa called Tajo. Somewhat less ambitious in its goals than Cedar, Tajo nevertheless included a number of EPE-related features: large virtual address space, emphasis on integration and consistency of user interface, support for concurrent operations, a uniform screen manager and window system, and a variety of packages and tools.

From 1978 to 1983, the Smalltalk group concentrated on taking Smalltalk to the world outside Xerox, both as an exercise in portability, and so that it could be shared and built on by other groups with similar views about system design. There were several aspects of this work: redesigning the system for available microprocessors, specifying and publishing a standard virtual machine definition, refining and licensing a portable standard virtual image of the Smalltalk programming environment, and publication of a series of books documenting the results of the Smalltalk work. This effort culminated in 1983 with the public release of the system in the series of Smalltalk-80 books from Addison Wesley, and licensing by the Xerox Corporation of the complete Smalltalk-80 programming environment. By the close of 1983, Smalltalk had been successfully ported to the DEC VAX, the Intel 8086, and several machines based on the Motorola 68000.

In the process of documenting and releasing the Smalltalk system, the system itself was rewritten almost from scratch. A number of EPE-related facilities were also developed during this period, including: a new "Model-View-Controller" framework that enabled separation of computational models from viewing mechanisms and became the basis of the entire user interface, a version management system, a document editor for text and graphics, and an integrated mail system. Browsing capabilities were enhanced with access to callers, callees and class inheritance paths. Interactive correction of common coding errors was also considerably simplified, further reducing development turnaround time.

Interlisp

During the same period, the bulk of the Interlisp effort was also devoted to non-EPE related activities, namely the transferring of Interlisp from a time-shared, teletype-oriented system for the PDP-10 to a display-oriented system for high-performance, personal computers. Interlisp was totally re-engineered during this process. All of the operating system functions provided by Tenex or TOPS-20, were rewritten in Interlisp, as well as device controllers and networking facilities. All of these capabilities were implemented in the Lisp language itself, so that the entire system was made significantly more portable and easier to maintain. The user interface to all of the existing Interlisp packages, such as Masterscope, the programmer's assistant, and the Interlisp editor, was redesigned to take advantage of the high-resolution display.

In addition, the following EPE-related facilities were also added to the system: a process mechanism, support for object oriented programming [3], a uniform screen manager and window system. a high-quality text editor combining graphics and text. an integrated mail system, and performance tools.

A Tour Through Cedar

The next paper in this report, "A Tour Through Cedar," shows the state of Cedar (as of September, 1983). It contains a highly visual presentation of the current Cedar system in the form of a demonstration of the system during which many of the components of Cedar discussed earlier, including Tioga, Viewers, the Userexec, and Walnut, are presented and discussed. A somewhat condensed version of "A Tour Through Cedar" appears in the Proceedings of the Seventh International Conference on Software Engineering, March, 1984, Orlando, Florida, as well as in the April 1984 issue of IEEE Software.

A Tour Through Cedar

Introduction

This paper introduces the reader to many of the salient features of the Cedar Programming Environment, a state-of-the-art programming system that combines in a single integrated environment: high-quality graphics, a sophisticated editor and document preparation facility, and a variety of tools for the programmer to use in the construction and debugging of his programs. The Cedar Programming Language is a strongly-typed, compiler-oriented language of the Pascal family. What is especially interesting about the Cedar project is that it is one of the few examples where an interactive, experimental programming environment has been built for this kind of language. In the past, such environments have been confined to dynamically typed languages like Lisp and Smalltalk.

The paper attempts to give the reader the feel of the Cedar system by simulating a live demonstration. The demonstration is actually taken from a video tape of such a live demo; the sequence of events, as well as the dialogue, is fairly close to what a viewer of this tape would see and hear. Numerous snapshots of the display taken at various points during the session simulate the visual information contained in the tape. Text that would actually appear on the display during a demonstration–either because the user typed it or the system printed it–will appear in this paper in a distinguished font. The explanations that the demonstrator would give will be in the normal font. Observations and comments that would be distracting during a live demonstration, but are appropriate for a paper, are included as footnotes.†26

Now let's begin our tour.

The Display

You are looking at (see Figure 1) a bitmap display connected to my personal computer, a Dorado.†27 The figures you see at the bottom of the screen in Figure 1 are called *icons*. They represent objects that are of potential interest, but not currently in active use. Some of them represent text documents, scanned images, or other data structures that I can look at and manipulate. Others represent tools or services that I can use. Their shapes are meant to be suggestive of their functions. For example, the icon on the lower right that looks like a mail box represents my mail reader, called Walnut. The fact that the flag on the mail box is up indicates that I have new mail. The icon next to the mailbox that looks like a stack of envelopes represents my active message set. We will use both of these later in the demonstration. The icon next to my messages is used for sending hardcopy to the printer whose name is Clover (located

†26 These footnotes contain a lot of information about Cedar: why we did things certain ways, how useful a particular feature turned out to be, etc. For some readers of this paper, the footnotes will contain the most interesting material. However, the reader who is unfamiliar with Cedar and simply wants to get an overview might find the footnotes distracting to the flow of the demonstration. Therefore, a good way for him to read this paper might be to ignore the footnotes on the first reading (especially the long ones), and then come back to them later.

†27 All Dorados use as a display a high-resolution television monitor, 1024 pixels wide by 808 high. The physical dimensions of the display are 12" x 9". Figures in this paper that show the entire display are about 1/2 scale (but full resolution). Dorados originally used a narrower monitor, 608 by 808, but we have found that for both editing and programming tasks the extra width was extremely desirable. In fact, many users feel that they could make effective use of even more screen real estate, and would like to be able to connect more than one monitor to the same machine. We think that this is feasible, both from the software and human engineering aspects; the hardware is certainly capable of supporting it. In fact, those researchers with applications involving color already operate in a configuration consisting of a wide-screen black-and-white display adjacent to a 1000-line color monitor, both connected to the same Dorado and maintained by the same software.

down the hall), and the icon in the left corner of the display that looks like a file cabinet views the FileTool, a facility for obtaining files from remote servers.†28

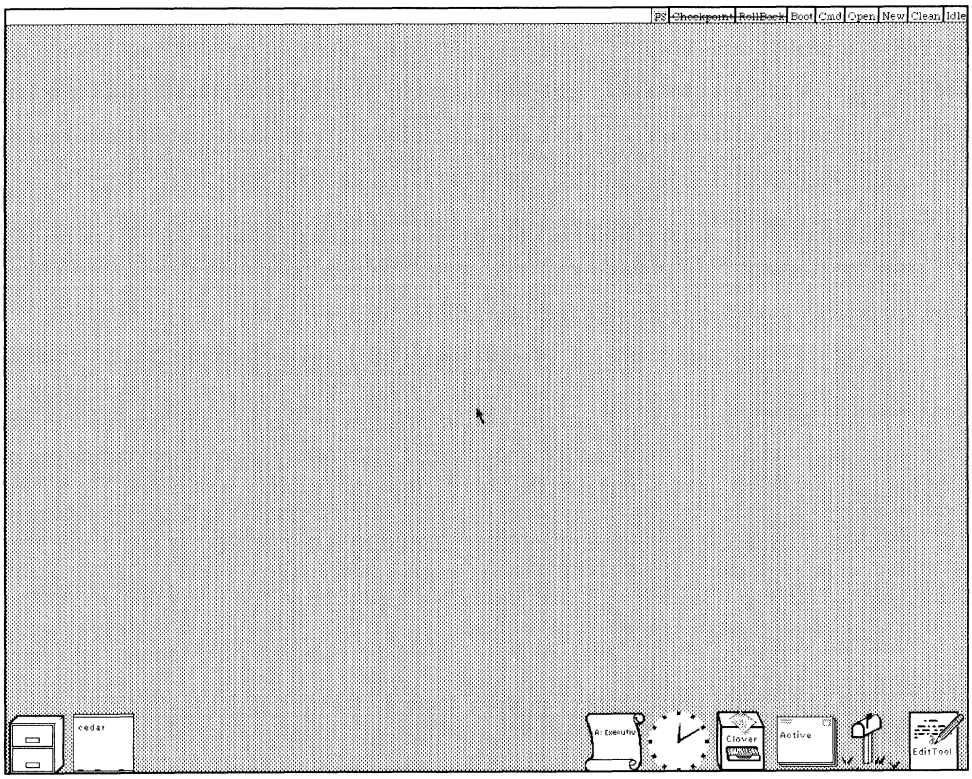


Figure 1

Initial Cedar screen layout showing various icons

Viewers Window Package

The Viewers Window Package provides the basic display paradigm for Cedar [19]. It allows users and programs to create, destroy, move, and resize rectangular individual viewing areas called *viewers*. (To a first approximation, a viewer corresponds to what is called a window in many other systems.) Some viewers present textual or graphical data to the user; others provide the user with various forms of control, such as access to facilities or the ability to invoke procedures. Viewers that provide access to a

†28 In a traditional time-sharing environment, users share files straightforwardly since all files reside in the same place. In our distributed environment, files that are created by a user on his personal machine can only be shared if they are stored on another machine called a *file-server*, a computer with a large disk dedicated to the task of storing and retrieving files, to which all of the personal machines have network access. For files that are part of the standard system, such as sources, documentation, and fonts, the user need not be aware of where the files are stored, or whether they have already been retrieved onto his local disk–the system takes care of this automatically for him using a *version map* that is built when the system is released. However, the user must explicitly store, retrieve, and keep track of files that are not part of the standard system (but there are packages to aid him in this task).

facility are called *tools*, and viewers that simply invoke a procedure are called *buttons*.†29 The FileTool and the Walnut Mail Reader shown at the bottom of Figure 1 are examples of tools, and the nine small boxes labeled Idle, Clean, New, et al, at the upper right in Figure 1 are examples of buttons.

The icons at the bottom of Figure 1 are also viewers, namely viewers in their iconic form. *Opening* an iconic viewer tells the Viewers Package to allocate screen real estate to the viewer in the center portion of the display (see Figure 2), thereby allowing the viewer to present its contents in a more detailed and comprehensive fashion. Conversely, *closing* a viewer releases the space that the viewer currently occupies, and causes it to be displayed in iconic form at the bottom of the screen.

The user can open an icon by pointing at it using a device called a mouse [32]. Pointing is accomplished by sliding the mouse along a horizontal surface to position a mouse-controlled cursor on the display. (In Figure 1, the cursor is displayed near the center of the screen as an arrow.) When the desired location is reached, the user depresses and releases one of the three buttons located on top of the mouse. We use the verb *click* to describe this act of positioning the cursor and pressing and releasing a button. Let's open the icon for the Clock and for the FileTool. This produces the configuration shown in Figure 2 in which both the Clock and the FileTool viewers now occupy large, rectangular areas whose height is nearly the height of the entire display.

Most top-level viewers (viewers that are themselves not contained as part of another viewer) include a collection of buttons for invoking various operations associated with that viewer. For example, the FileTool viewer includes buttons for retrieving, storing, and listing files. The user clicks a button to make the corresponding operation happen. Often, these buttons are arranged in a horizontal array called a *menu* that is displayed just below the viewer's *caption*, the black area at the top of each opened viewer that contains the viewer's name. For example, if you look at Figure 2, you will see that the Clock has a menu that includes the buttons SwapColor and ChangeOffset.†30 More elaborate menus are associated with text viewers, as shown in Figure 7.

†29 The principal difference between tools and buttons is in the number of operations and degrees of freedom they provide to the user. Tools typically allow the user to specify a number of parameters (and retain these parameters between invocations), whereas a button may take an argument, but essentially performs the same operation each time.

†30 The Viewers Window Package directly supports the horizontal arrangement of buttons into menus and the positioning of such menus below the caption; the implementor simply calls a procedure specifying a viewer and a button, and the Viewers Package does the rest. Since this is so convenient, most simple, data-presentation viewers use this facility when providing buttons. For example, the Cedar Documentation Browser shown in Figure 5 provides a menu including the buttons Reset, Freeze, NewBox. For viewers that represent tools, such as the FileTool shown here in Figure 2 and the WatchTool shown in Figure 15, each viewer typically provides access to a different and unique service, requiring a different display interface. Thus, the implementor of a new tool has to design and implement the display interface that seems appropriate for that particular task. Since the implementor is explicitly specifying the display of the viewer anyway, he often will explicitly position within the viewer the buttons provided by the tool, rather than using the default arrangement of menus below the caption.

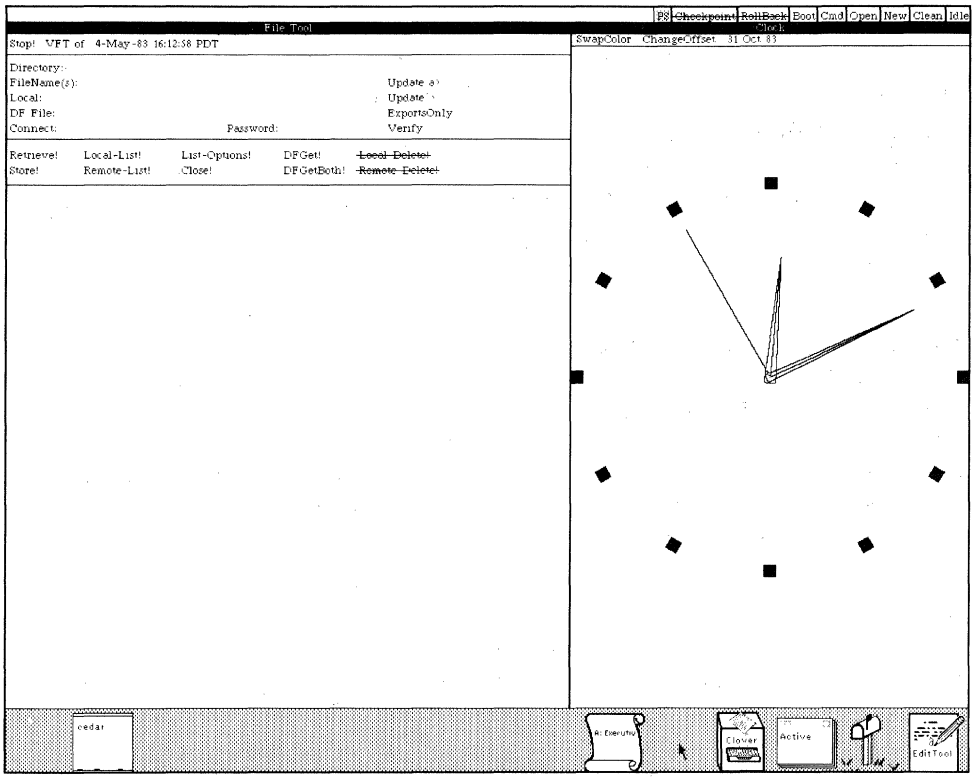


Figure 2

Same screen layout after opening the FileTool and Clock viewers

In addition to buttons specific to particular classes of viewers, buttons for various operations that apply to *all* viewers regardless of their class, such as Destroy, Close, and Switch columns, are contained in a menu that is hidden under the caption. This caption menu is only displayed when the mouse is actually in the caption area (it can be seen in Figures 4 and 11). Other buttons for invoking system-wide activities, such as creating a new viewer, performing a checkpoint, and booting, are not contained in a particular viewer but instead are included in the message area at the top of the screen (see Figure 2). For example, the button PS (PrintScreen) is used to produce hardcopy images of portions of the screen and was used to generate the figures in this paper. The remainder of the message area is used for displaying various comments about the system's status and behavior. The bottom portion of the screen is used for displaying icons.

The large, middle part of the screen that, in Figure 2, is occupied by the FileTool and Clock viewers, is divided into two columns.†31 When more than one viewer is created or opened in the same column, the viewers automatically share the available space. Conversely, when a viewer is closed or destroyed, the screen space that it occupied is then shared among the remaining viewers in its column. If a viewer is *grown*, i.e., given the full column to itself, then any other viewers in that column are automatically closed.†32 To show you how this works, I'll open the remaining icon on the left side of Figure 2, the one labeled "Cedar" that looks like a chalkboard with erasers on its ledge. This produces the arrangement shown in Figure 3.

†31 Both the width and height of these columns can be easily adjusted by the user using the mouse.

†32 This strategy of placing viewers adjacent to one another with no overlapping and no blank space is called *tiling* the screen. It is one of the most widely discussed aspects of the Cedar user interface, and often leads to heated, religious debates between its adherents and advocates of overlapping windows such as those employed in Interlisp and Smalltalk. However, regardless of how they resolve them, each of these screen management systems deal with the following issues: (a) provide for some form of default window placement so that the user does not have to be involved in specifying the position and size of windows if he does not wish to; (b) allow the user flexibility with regard to screen layout (in particular, some way of overriding default window placement); (c) strive to make maximal use of the screen real estate; (d) give the user a predictable and intuitive model about what will happen to the display when he performs a given operation. With regard to this framework, the two screen management algorithms have different advantages and disadvantages. For example, overlapping windows give the user a lot of flexibility with regard to screen layout, but can lead to wasted, i.e., unused, screen space. Also, the extra degrees of freedom provided by overlapping windows require that the user (or program) must specify additional information in placing a window. On the other hand, overlapping windows are more economical in that no window need be larger than the information it contains. Overlapping windows also have the advantage that the working set of active windows can be quite large, since only a small portion of a window has to be visible for the user to have access to the window. (This effect of using the corners as *handles* for those windows that the user might want access to is provided for in Cedar through icons.). However, users wind up spending a fair amount of time ensuring that the desired corners are always visible, and even so, overlapping windows seem to have an uncanny knack for getting lost. Something that is sometimes good and sometimes bad about overlapping windows is that changing the size or position of one does not change any of the others.

Similar arguments can be made about various schemes for menus. Pop-up menus are more economical with screen real estate as compared to fixed menus: they require screen space only when they are being used. However, pop-up menus do not allow the user to know ahead of time exactly where the menu will appear, and therefore require visual feedback before the operation can be completed. Pop-up menus also do not permit buttoning ahead, which can be annoying when system response falls behind the user, because it requires the user to wait for the system to catch up before he can input the corresponding operation. (The hidden caption menu in Cedar is a form of pop-up menu. When it was first introduced, users complained about the need to scan the menu to find the desired button, and the inability to button ahead. In response to their complaints, the two most common operations, Grow and Close, can now be invoked by simply clicking anywhere in the caption area with the middle or right mouse button respectively. Thus, the user does not have to wait for the hidden menu to be displayed and then visually find the Grow and Close buttons.)

Thus, there is room for considerable disagreement. It is, however, interesting to note that Interlisp is beginning to experiment with icons, while the use of pop-up menus is being discussed for Cedar.

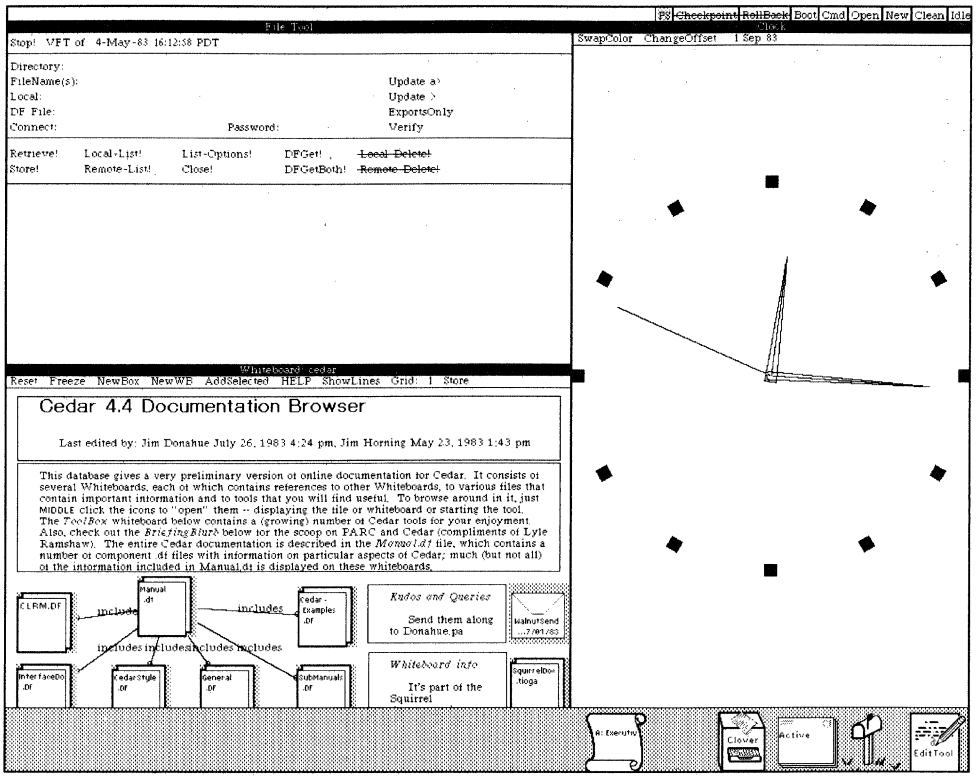


Figure 3

Viewers for the FileTool and Documentation Browser share the left column

Whiteboards

The viewer that I just opened is an example of a class of viewers called *whiteboards.* A whiteboard is simply a viewer consisting of a two-dimensional area in which viewers and text can be inserted, removed, or repositioned.†33 The whiteboard at the bottom of the left column in Figure 3 serves as a documentation browser for Cedar. Notice that not all of the information on the whiteboard is visible in the viewer; the bottom of the viewer clips off additional information. This particular class of viewers, whiteboards, elects to simply clip information that is not visible, rather than scaling the display to fit the amount of screen space available, as the Clock does in Figure 5 (upper right).

†33 Whiteboard viewers attempt to provide a spatial way of organizing data. They resulted from one person's observation that he liked to arrange material on his desk spatially, and could usually remember *where* he placed something more easily than remembering under what category he filed it in a filing cabinet, or the name of the electronic file in which he stored it. Whiteboards appeared in Cedar the following week, courtesy of John Maxwell.

In order to see more of this whiteboard viewer, let's move the FileTool from the left column to the right column using the appropriate button contained in the menu that is hidden under the FileTool's caption. We do this by moving the mouse into the caption area of the FileTool, which causes the caption menu to be displayed as shown in Figure 4 (the bullseye shape in the menu is the cursor), and then clicking the button labeled "-->". This produces the screen layout shown in Figure 5.†34

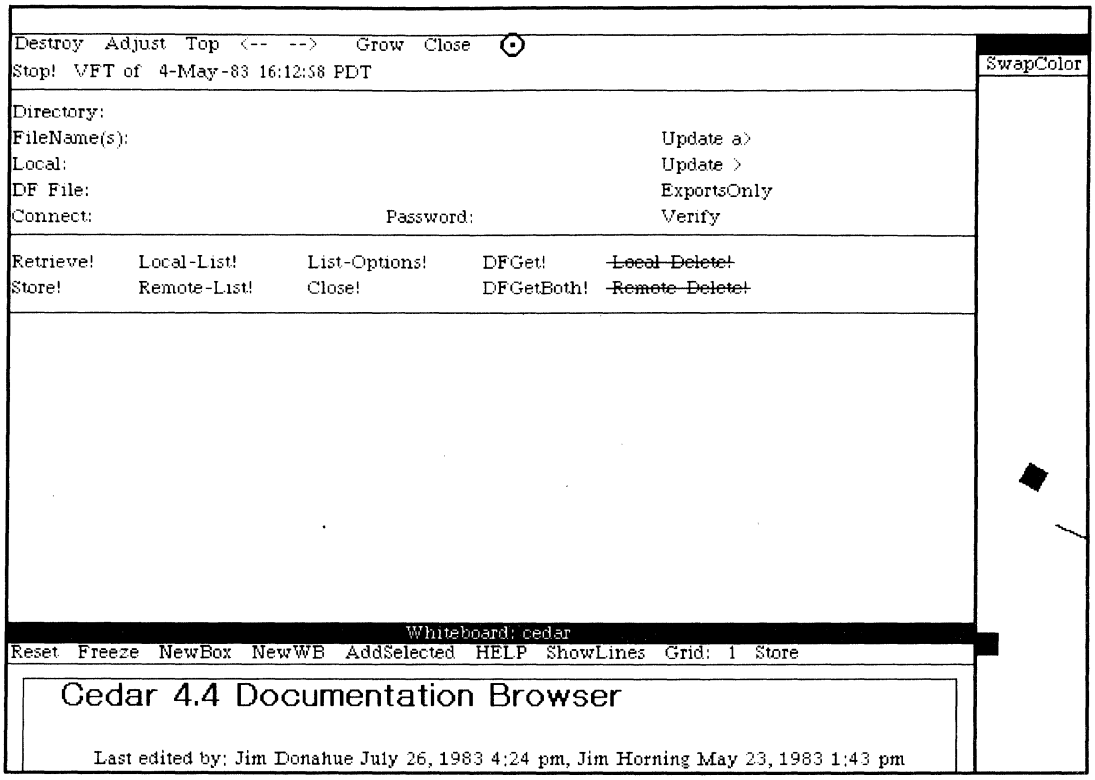


Figure 4

The caption menu becomes visible when the mouse is moved into the caption area

†34 Note that we could have accomplished the same result by growing the whiteboard using the Grow menu button in the same caption menu to give the whiteboard the whole column. Or, we could have simply closed the FileTool, which would also have caused the whiteboard to get the entire column, since it would have been the only viewer that remained in that column. Alternatively, without changing the current arrangement of viewers, we could have viewed other parts of the whiteboard viewer by simply scrolling it, the same as we would a text document.

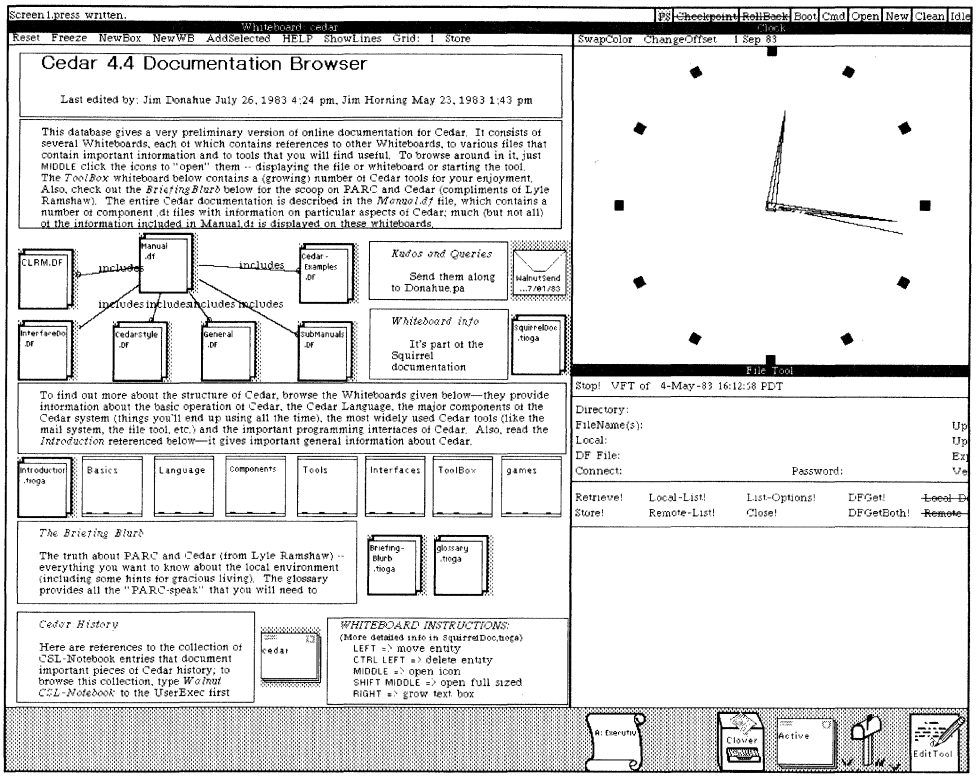


Figure 5

The Cedar documentation browser

Online Documentation

The Cedar Documentation Browser shown in Figure 5 uses a whiteboard viewer to display a data base for the online documentation for Cedar [5]. About halfway down this whiteboard is a row of icons for seven other whiteboards: Basics, Language, Components, Tools, Interfaces, ToolBox, and Games. We can find out more about any of these aspects of Cedar by browsing the corresponding whiteboard. To do this, we follow the instructions displayed in the lower right corner of the whiteboard: we move the mouse into the corresponding icon and click the middle button. This will cause a new viewer for the corresponding whiteboard to be created and displayed.†35 For example, let's open the Components whiteboard, which includes whiteboards for various important components of Cedar such as the Viewers Package, the Tioga editor, and the UserExecutive. The Components whiteboard in turn contains an icon for the Viewers Package whiteboard. If we middle-click this latter icon, we get the configuration shown in Figure 6.†36

†35 The system will automatically obtain the necessary information from the corresponding data base, which is stored on a file-server. All of this happens reasonably quickly (a few seconds).

†36 In order to obtain the screen layout shown in Figure 6, I also moved the Viewers whiteboard to the right column, made it full size, and slightly readjusted the width of the columns. Having already explained how viewers are moved and changed, I will refrain from itemizing each and every such operation for the remainder of the paper.

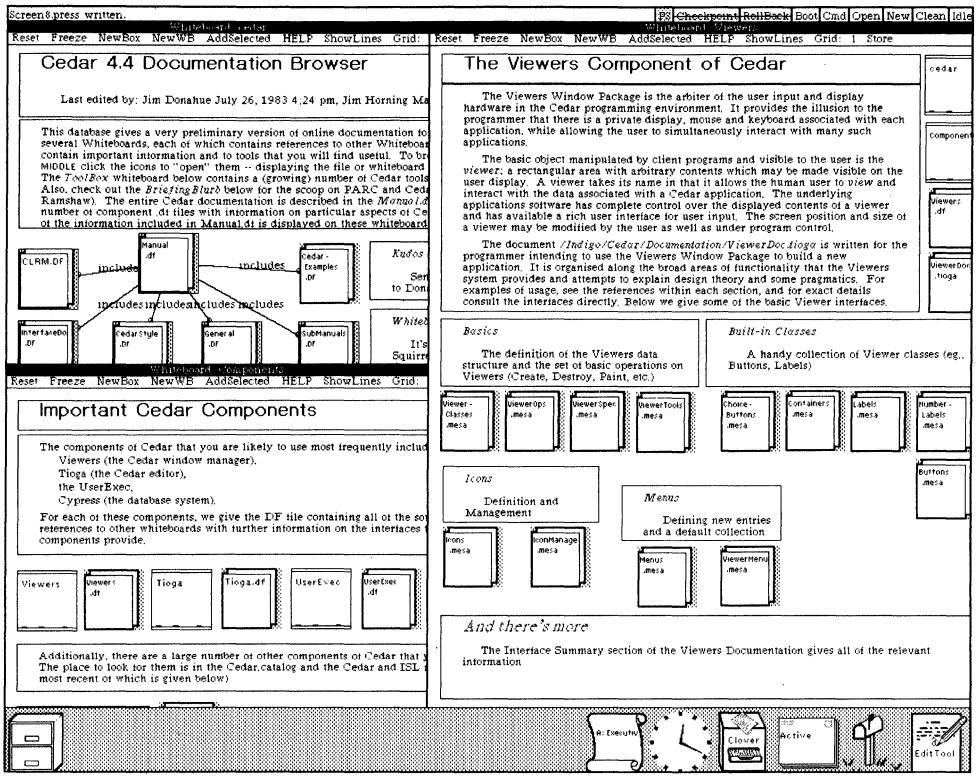


Figure 6

Browsing the documentation using whiteboards

The whiteboard for the Viewers Package that appears on the right in Figure 6 includes icons for the various public interfaces of the Viewers Package, as well as an icon for the Viewers Package online documentation contained in the file ViewerDoc.Tioga. We can cause this documentation to be displayed using the same method as we did to display the whiteboards, namely by simply moving the cursor into the icon and clicking.†37

†37 We have placed a great deal of emphasis in the design of Cedar on uniformity of command interface. In fact, "Uniformity in Command Interface" was one of the items in our catalogue of desired programming environment capabilities (see Appendix 2 of this report). "What is important about a standard user interface package is that the user be able to confidently predict the general manner of interaction with a program that uses the package, even though he hasn't experienced it yet; and that by and large, the user will be right. This has been called the Law of Least Astonishment" [8]. Here is a testimonial from a new user regarding the user interface in Cedar taken from an electronic message sent to the Cedar implementors (the user has used the word "integration" where the author would have used "consistency" or "uniformity"): "I can't praise you all enough for the way that integration helps a beginner when he encounters it. I've started using Clean and wanted to throw away some stuff from my MRU cache. How to do that? Well, what works other places? Control ... control-left-click. Got it in one. Ahhh, wonderful. Novice user feels smart, happy, wishes to learn more. More integration, please."

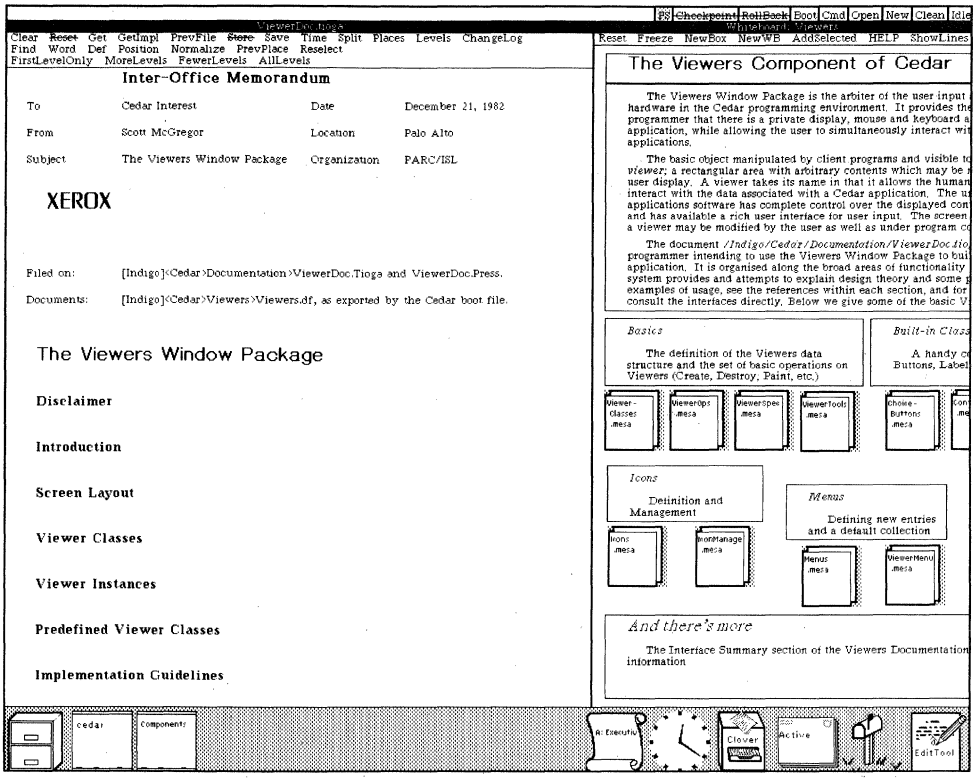


Figure 7

Online documentation for the Viewers Package

The Tioga Editor and Document Preparation System

The text viewer that appears in the left column of the display in Figure 7 is the on-line documentation for the Viewers Package itself, in the form of a Tioga document. Tioga is both the editor for Cedar programs as well as its document preparation system.†38 †39 In Tioga, a document is a tree

†38 It is worth pointing out that the Viewers Package documentation shown in Figure 7, as well as all Cedar documentation, was prepared using the Tioga editor, as was the paper that you are now reading. When hardcopy is needed, the Tioga typesetter (represented by the printer icon shown at the bottom of Figure 7, fourth icon from the right) is used to generate high-quality hardcopy from the document and send it to the corresponding printer.

†39 In many environments, document production systems are frequently de-coupled from text editors, e.g., Scribe, TEX, Pub. "One normally takes the text that one wants to include in a document, wraps it in mysterious commands understood by a document processor, feeds it to that processor, and puzzles over the resulting jumble of characters on the page. In short, one programs in the document processor's language using conventional programming tools–an editor, a compiler, and sometimes even a debugger. Programmers tend to think this is neat; after all, one can do anything with a sufficiently powerful programming language. However, document processors of this sort frequently define bizarre and semantically complex languages, and one soon discovers that all of the time goes into the edit/compile/debug cycle, not into careful prose composition" [23].

At PARC, we favor the WYSIWYG (pronounced whiz-ee-wig) approach to document preparation systems. WYSIWYG is an acronym for What You See (on the screen) Is What You Get (on paper). A single program provides both the usual editing functions and a reasonable collection of formatting tools. You can't program a WYSIWYG editor as you would a document compiler, but you can get very tolerable results in far less time.

Strictly speaking, Tioga is not a WYSIWYG editor; there are various operations performed by the typesetter in producing hardcopy that are not faithfully reproduced on the screen, such as filling and justification. (Mostly, this is a performance issue.) However, Tioga certainly inherits the spirit of WYSIWYG editors.

structure of nodes rather than a list of paragraphs so that a hierarchical structure can be explicitly represented. Successive levels correspond to greater levels of detail, and the viewer of a Tioga document can be instructed to suppress the display of all nodes deeper than a certain level. For example, in Figure 7 only the top level of nodes are shown, thus effectively providing a table of contents.

In combination with scrolling, the use of levels in Tioga makes it easy for the user to browse through a document or program source and quickly find the part that interests him. For example, let's scroll to the section entitled "PreDefinedViewer Classes"†40 and click the MoreLevels menu button at the top of the viewer.†41 This allows us to see one more level of detail, the titles of subsections, as shown in Figure 8.

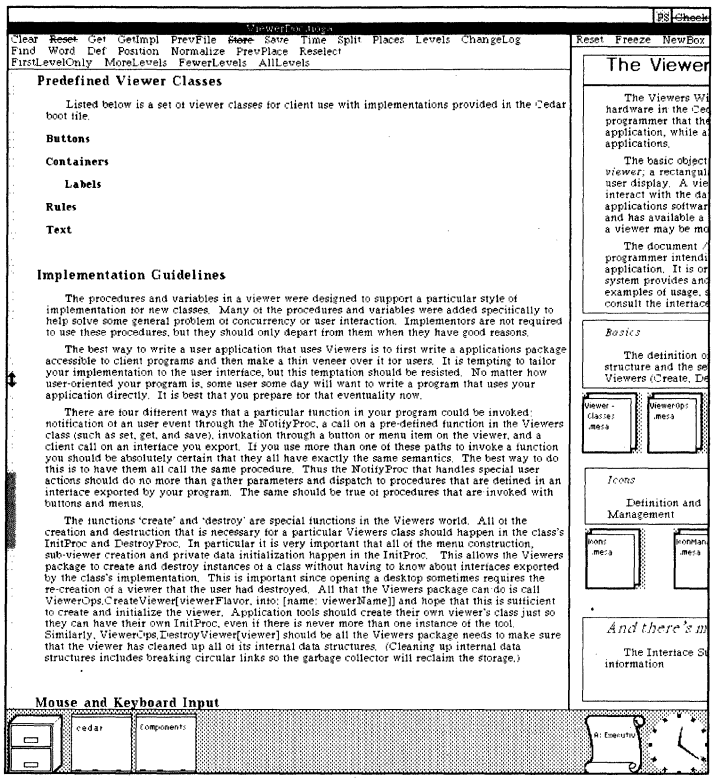


Figure 8

Browsing a Tioga document using level clipping to suppress detail

---

†40 Scrolling is accomplished by moving the mouse into the *scrollbar*, a vertical area at the left side of a viewer, and then clicking the mouse. The cursor (the double vertical arrow) is in the scrollbar in Figure 8, causing it to be displayed as a grey bar. The darker part of the scrollbar represents the part of the document that is currently visible. The user scrolls the viewer up by an amount equal to the distance from the mouse to the top of the viewer by clicking the left button on the mouse, and scrolls down a like distance by clicking the right button. Clicking the middle button scrolls the viewer by an amount proportional to the position of the cursor in the viewer. For example, if the cursor is 1/3 of the way down from the top of the viewer, scroll to 1/3 of the way from the beginning of the document.

†41 The more advanced user can perform this same operation with a single action by holding down the SHIFT key while scrolling. This is an example of our concern for an efficient interface for experts. Many systems that boast of being extremely easy to use have the drawback that they do not allow the experienced user to become much more proficient with the system than the novice user. For experts, the desire for common operations to require a minimum of effort can be more important than the desire for the greatest possible simplicity in the user interface. However, in order to protect novice users from accidentally invoking an esoteric operation and becoming confused, each user of Cedar specifies in his user profile his *user category*: Beginner, Intermediate, or Advanced. For users that are Beginner or Intermediate, certain commands and operations are disabled.

Clicking the MoreLevels menu button again would show yet further detail, i.e., the contents of the subsections entitled Buttons, Containers, etc. Now let's scroll back to the beginning of the document and I'll briefly demonstrate how the Tioga editor works.

The Tioga editor allows the user to select individual characters, words, or entire nodes or branches (a branch is a node plus all of its children). For example, I can select the word "environment" in the first paragraph of the introduction (see Figure 9) by pointing at it and clicking the middle button of the mouse. This does two things. First, it establishes the *input focus*, i.e., tells the Viewers Package that any characters that I type should be seen and interpreted by this viewer, not by some other viewer also waiting for input. Secondly, clicking the mouse in a Tioga document tells the Tioga editor the location of the current *insertion point*, in this case, immediately following the word "environment."†42 Tioga indicates the current insertion point on the display by the appearance of a blinking caret. (The caret can be seen in the third line of the first paragraph, just after the word "over.") Basically, what all this means is that to use Tioga, you simply point and type and the characters are inserted into the document at the place where you pointed. Figure 9 shows the state of this document after I pointed at the word "environment" in the second line of the first paragraph and typed "Here I am in the process of inserting material: the quick brown fox jumps over."

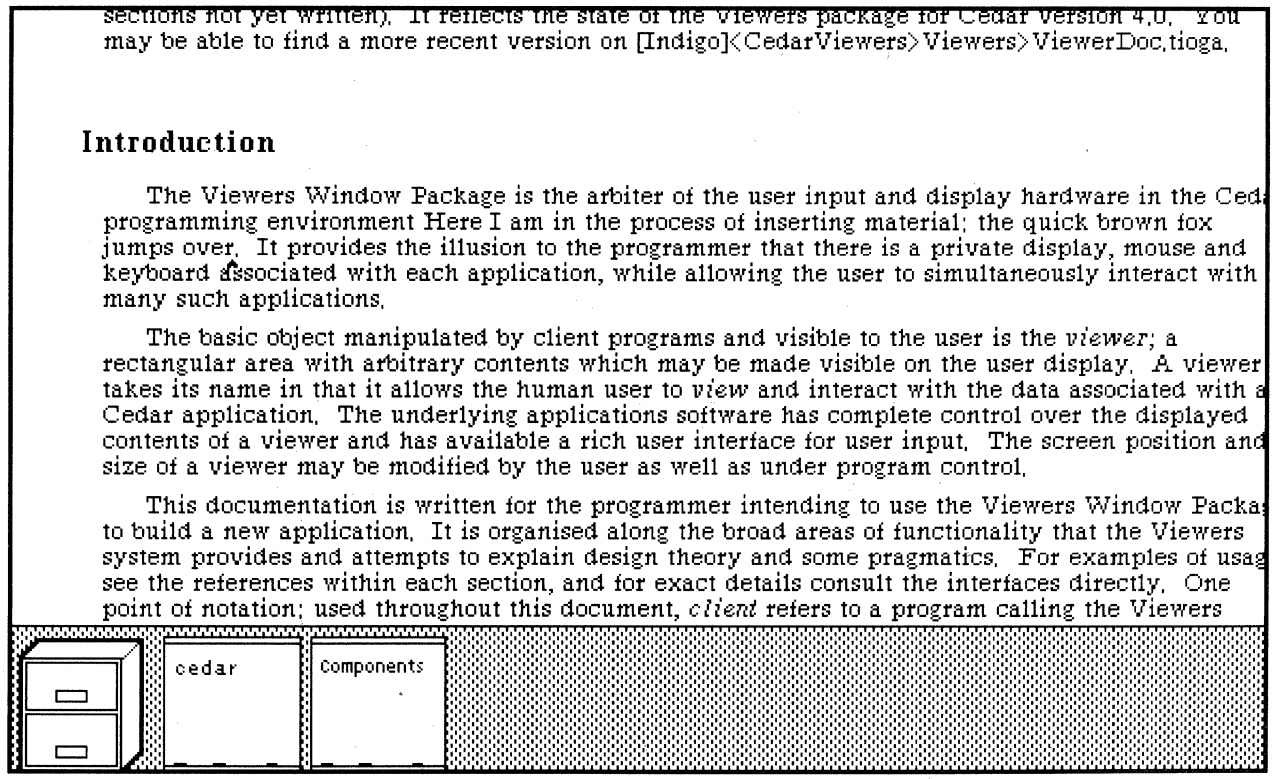


Figure 9

Inserting characters into a Tioga document

†42 When the user selects the space between two characters, the insertion point is unambiguous. For other selections, e.g., character, word, node, etc., the insertion point is taken as the end of the selection closest to the actual position of the mouse when the selection was made. For example, if I selected the word "environment" by pointing at the "o" or any character to its left, the insertion point would be at the beginning of the word, whereas if I selected the second "n" or any character to its right, the insertion point would be at the end of the word. This arrangement is fairly intuitive and works out quite well in practice.

Commands can be given to Tioga using various control keys, e.g., typing a character while the CTRL key is depressed.†43 For example, I'll *undo* the insertion I just made with a single keystroke. This ability to undo arbitrary editing operations allows the user to recover from mistakes.

Another command that I can give to Tioga is to change the way characters appear by changing their *looks*.†44 For example, let me emphasize a sentence of this document to draw it to your attention by making it appear in a larger font and underlined (as shown in Figure 10).†45

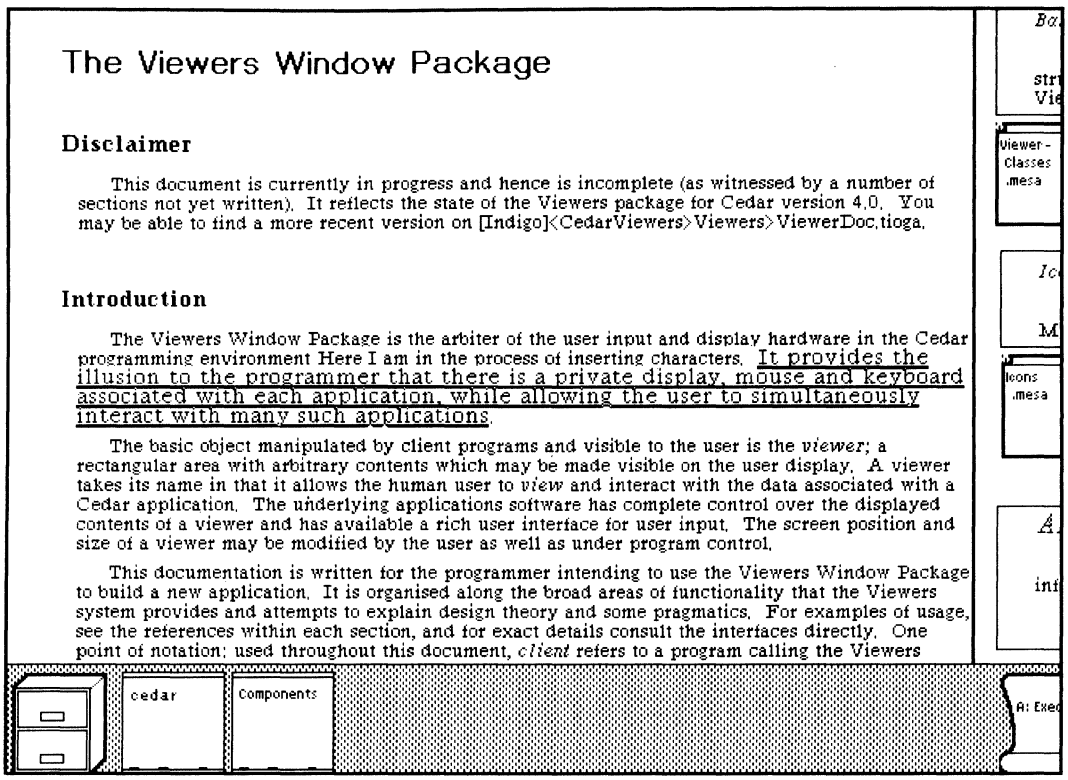


Figure 10

Changing fonts

†43 Commands can also be given to Tioga via the EditTool, the right-most icon in Figure 7. The EditTool is menu driven and, as such, is self documenting; if the user does not know the particular key and mouse combinations to invoke an infrequently used operation, it is still an easy matter to perform it using the EditTool. The EditTool also provides the user with some additional, powerful capabilities not available through the keyboard, such as the ability to specify fairly complicated search operations and patterns, as well as to construct sophisticated macros.

†44 The documentation for Tioga explains its underlying structure as follows: "Each node in a Tioga document contains text. The characters of the text can have *looks* which control various aspects of their appearance such as font and size. Appearance is also influenced by the *format* of the node which determines things such as vertical and horizontal spacing. The document contains names of looks and formats, but not the specific interpretation of them. The interpretations are instead collected in a *style* which can be shared by many documents. [For example, in the style for this paper, there are definitions of formats for headings, quotations, and program output, and similarly, there are definitions of looks for emphasis and for small caps.] Rather than copying the specific details for the formats and looks, the document refers to them by name so it is easy to change the definitions in the style and modify the appearance uniformly throughout the document" [22].

†45 Notice that to accomplish this font change, I do not insert commands into the document to change the font, but give the command directly to the editor, and see the result of the command immediately take effect.

The sentence that I underlined makes an important point: users can and do make heavy use of parallelism in Cedar. It enables them to start one task before another has finished, and to switch back and forth among several tasks, such as editing, compiling, reading mail, etc.†46

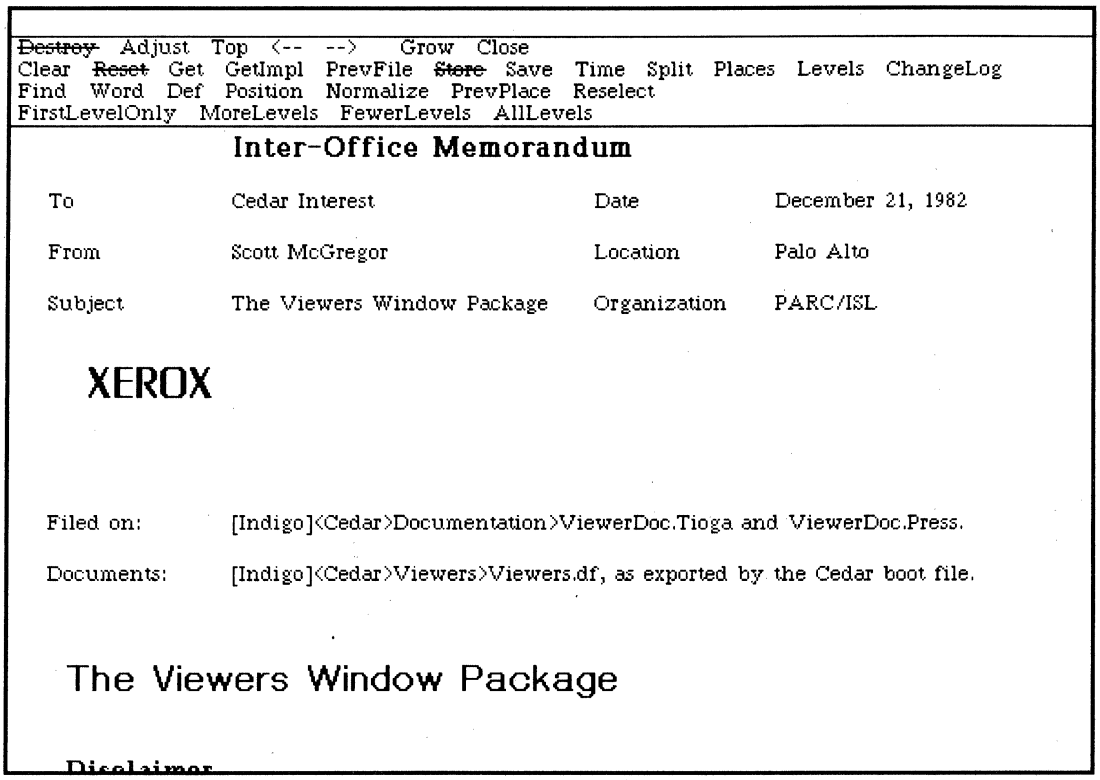


Figure 11

The Destroy menu button is guarded to prevent accidents

We aren't going to be needing this viewer, so let's destroy it using the Destroy menu button which is also contained in the caption menu. Notice that the Destroy button in the caption menu in Figure 11 has a line through it (whereas the Destroy menu button in Figure 4 does not). This indicates that the button is *guarded.* Guarded buttons must be clicked twice in a short time interval to take effect.†47 This

†46 To facilitate this parallelism, we have pursued in the design of the Cedar user interface what might be called the Principle of Non-Preemption: "Individual interactive programs operate in a non-intrusive manner with respect to the user's activities. The system does not usurp the attention and prerogatives of the user. A program responds to the user's stimuli, but then quietly retains its context and logical state until the user elects to interact with the program again, not (for example) monopolizing the resources of the computer" [8]. This is especially important in an environment such as ours where the use of personal machines encourages (and makes socially acceptable) using the time when the user is thinking or the time between keystrokes for performing various background processing, e.g., sending and receiving mail, printing, recompilation, and database maintenance. Such activity loses a lot of its utility and attractiveness if the user is continually forced to deal with unexpected interrupts from these background tasks.

†47 The first click removes the guard. If a second click does not occur within a specified interval (about five seconds), the guard is restored. We feel that this interface is preferable to having the system enter into a confirmation mode; the latter would violate our principal of non-preemption.

is to guard against the user's inadvertent destruction of useful work. For example, the first time any edits are made to a Tioga document, the Destroy button automatically becomes guarded. Similarly, the Local-Delete and Remote-Delete buttons in the FileTool (see Figure 2) are also guarded. Let's go ahead and destroy the viewer in Figure 11 anyway and see what happens.

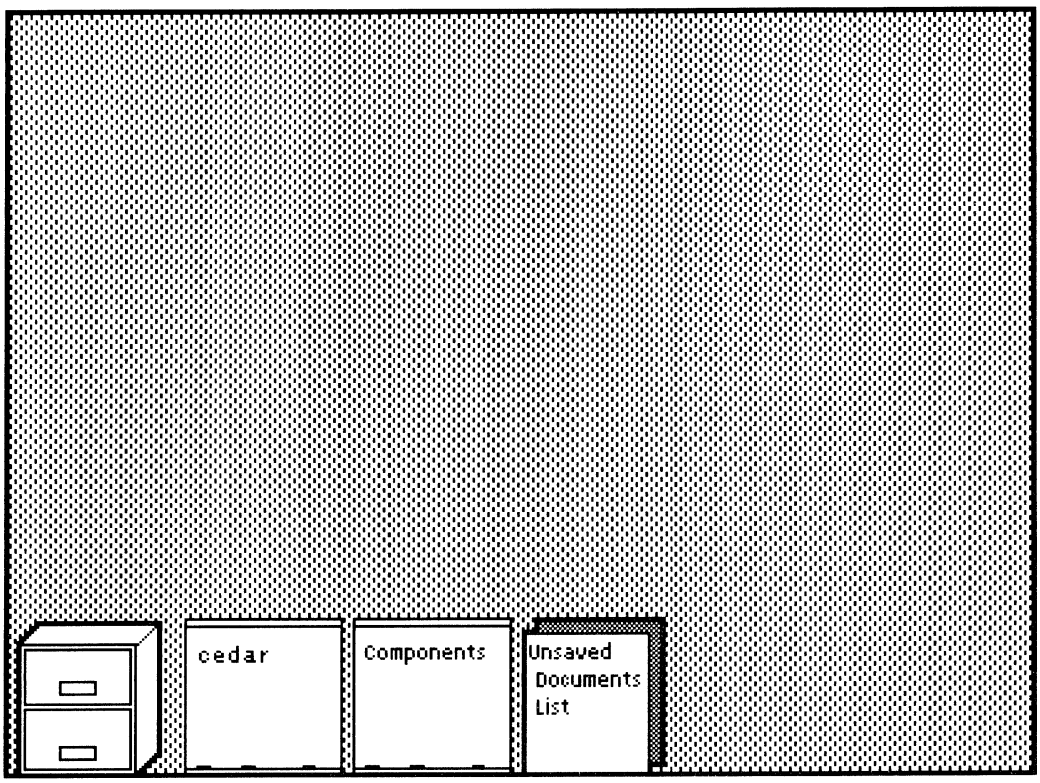


Figure 12

Recovering lost edits

In the lower portion of the screen, a new icon labeled UnsavedDocuments List has been created. If I were to open this icon, it would be found to contain: "The following files were edited but not saved. They may still be restored with edits intact simply by loading them. If you really want to get rid of the edits, load the file and hit Reset," i.e., I can still get my edits back if I really want them.

Such touches as undoing, guarded buttons, and the ability to recover destroyed edits, are what some might describe as frills. However, we believe that they contribute a surprising amount to programmer productivity. They allow the user to move ahead quickly with the confidence that he will be able either to avoid disaster or to recover from it. We have placed a great deal of emphasis on them in the design of Cedar.

The UserExecutive

An increasing number of users of Cedar are non-programmers; they use Cedar to prepare documents and read and send mail. However, Cedar is primarily a programming environment. So let us now focus our attention on the programming aspect of Cedar. To do this, I'll open up a UserExecutive. Notice that I said *a* UserExecutive, not *the* UserExecutive. Consistent with our philosophy of providing parallelism, there can be several instances of the executive, each with its own state, and performing its own operations.

Each instance of an executive is associated with a viewer called a Work Area through which the user interacts with the executive. Commands are typed to the executive by typing to this viewer, and output from the executive is displayed in the same viewer. At this point in the demonstration, there is only one instance of the executive; it is associated with the icon at the lower right of Figure 7 that looks like a scroll and is labeled "A: Executive." I'll open this icon in the usual way, and then move the mouse into the resulting viewer and click it. In Figure 13, the caret is in Work Area A indicating that this UserExec is now listening to me, i.e., it will see the characters that I type.

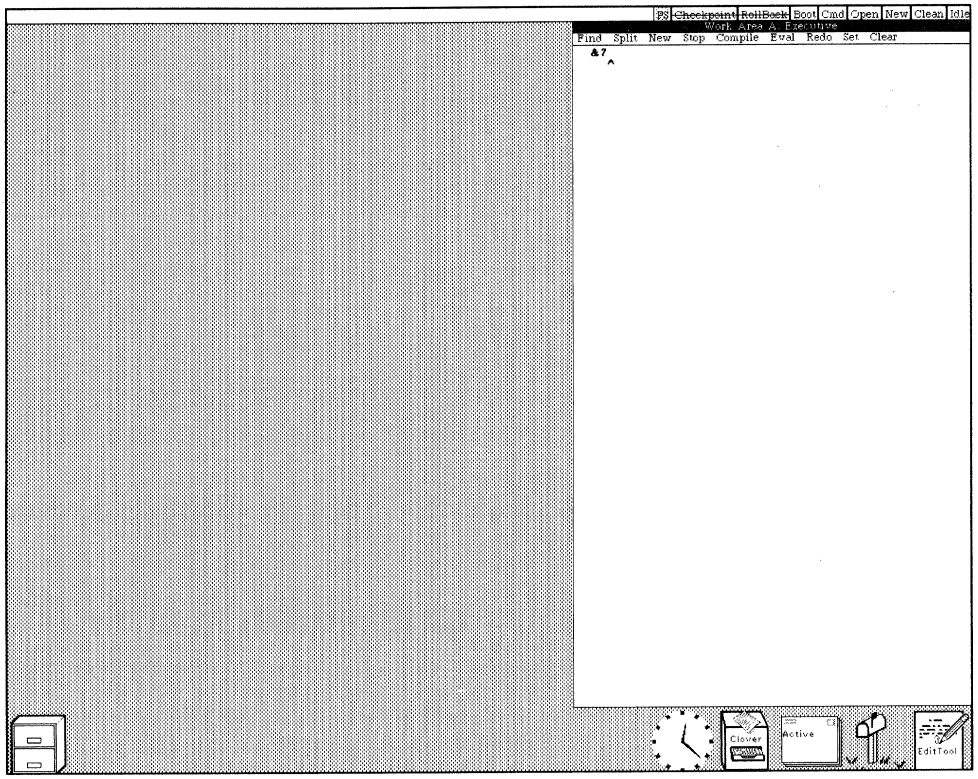


Figure 13

A UserExecutive waiting for commands

The Cedar UserExecutive implements various standard executive functions such as accessing the directory system, compiling, binding, loading, and running programs. Each interaction with the UserExecutive is called an *event*, and consists of a command name, followed by any parameters. The user can request explanatory information about a command or its arguments by typing "?". For example,

&7 run?
Run Load and Start the named programs. †48

†48 Text that actually appears on the display, either because the user typed it or the system printed it, will be in this font. The reason this event is number 7 is that it was preceded by six events consisting of various initializations to prepare for making the figures for this paper. The Work Area has been scrolled so that these events are not visible since they are not of interest.

The "?" indicates that I want to see more information about the preceding subject, in this case, the run command. The UserExec tells me that this command is used for loading and starting programs. I'll use the run command to run the program Watch, which is a performance monitoring tool that periodically samples and displays the words allocated, cpu load, and page faults.

```
&8 run watcch
watcch -> watch
Loaded and started: watch.bcd
```

I misspelled the name of the program to be run. In most systems, this would cause some sort of a FileNotFound error to occur. Instead, the Cedar spelling corrector was invoked, and given the name "watcch" and the context "a file to be run," quickly (a few seconds) produced a file which was reasonably close in spelling. The executive then loaded and started the corresponding program, which created the Watch tool box icon shown at the bottom of Figure 14.†49

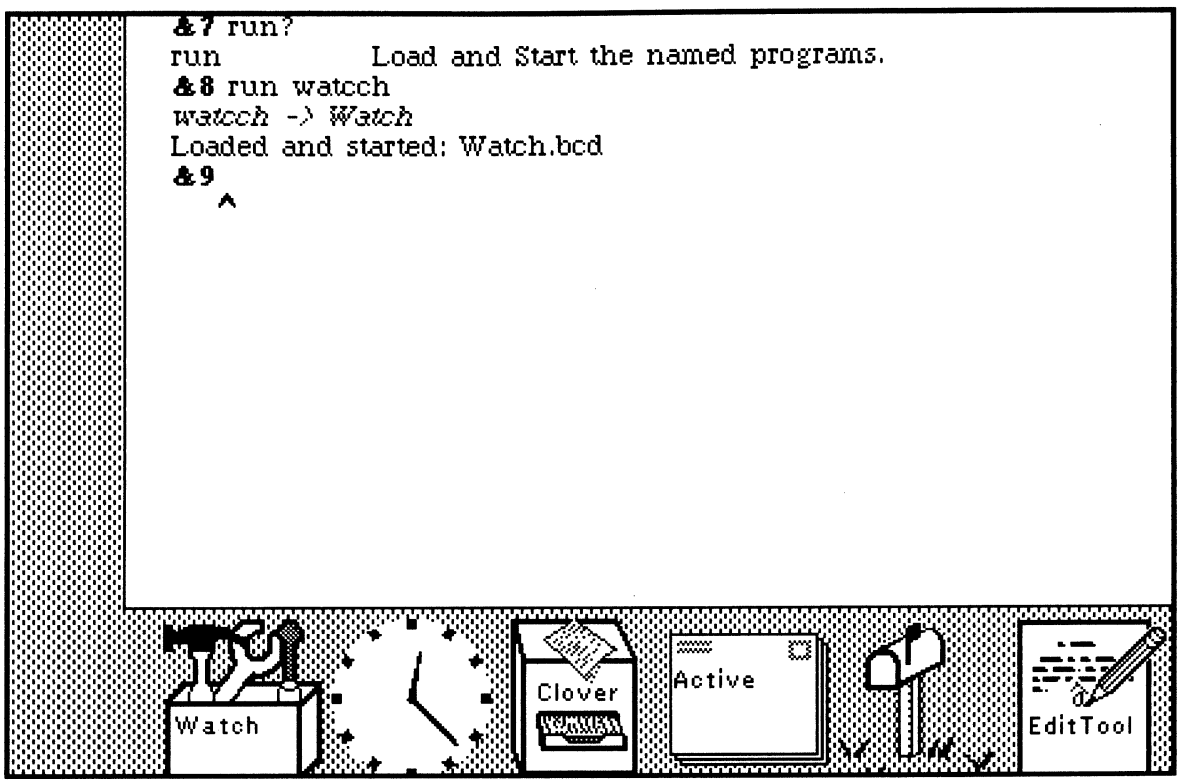


Figure 14

Loading and starting the Watch program

†49 Some tools supply their own icon, such as the FileTool and the TypeSetter (the printer icon). The toolbox icon employed by the Watch tool is provided as the default icon for those tools that do not. Not every implementor is artistic enough to design his own icon, though icon art is flourishing (some would say getting out of hand!).

The automatic correction of "watcch" to "watch" is an example of what we call DWIM, short for Do-What-I-Mean. The Cedar DWIM facility is patterned after the Interlisp DWIM facility in philosophy and style [31].†50

†50 DWIM is an important part of Interlisp. In fact, is has been cited as one of the most impressive features in the Interlisp system [25]. Nevertheless, the reaction of the Cedar user community to DWIM has been mixed, and serves to highlight some of the basic differences in style and philosophy between the Lisp and Mesa communities (most Cedar users come from the Mesa community). Historically, Mesa programmers have never really had the opportunity to develop their programs interactively in the sense that Lisp programmers do: when a change is required in a Mesa program, the programmer has to edit his source, compile it, correct syntactic errors (except for minor changes, it is unusual for a program to compile successfully on the first attempt), recompile, and then reload the program, before he can evaluate the effects of his change. For programs that change the system in some global way so that multiple instances of the same program cannot be simply loaded on top of one another, the programmer may even have to reboot (reload the system), or at least rollback to a previously established checkpoint, before he can load his program. Running on Dorados, we have been able to reduce this turnaround time to the order of minutes, rather than hours (or large fractions thereof), as was often the case in the Alto Mesa world. Nevertheless, the situation is still qualitatively very different from that of the Lisp programmer who can make a change and see the effect of his change immediately. Furthermore, as Erik Sandewall has observed [25]: "The average Lisp user often writes a program as a programming experiment, i.e., in order to develop the understanding of some task, rather than in expectation of production use of the program. The act of developing the program, not the act of running it, constitutes the experiment" [25]. During the course of such an experiment, the Lisp programmer expects to have to change his mind and his program many times. The Interlisp system also provides a general *undo* capability for allowing the user to reverse the effects of changes that he makes.

As a result of these factors, the average Mesa programmer tends to put more thought and planning into each change, and to proceed at a slower, more deliberate pace in interacting with the system when compared with the average Lisp user. Because of this philosophy of "go slowly, don't make mistakes because they are expensive to correct," Mesa users tend to make fewer careless errors when interacting with the system than Lisp users, so that the utility of DWIM is correspondingly reduced. However, the general consensus is that a DWIM facility for dealing with simple syntactic errors detected by the compiler would be well received by Cedar users.

I'll open the Watch icon, and then we'll observe the Watch tool in action as I execute another event in the UserExec (see Figure 15).†51

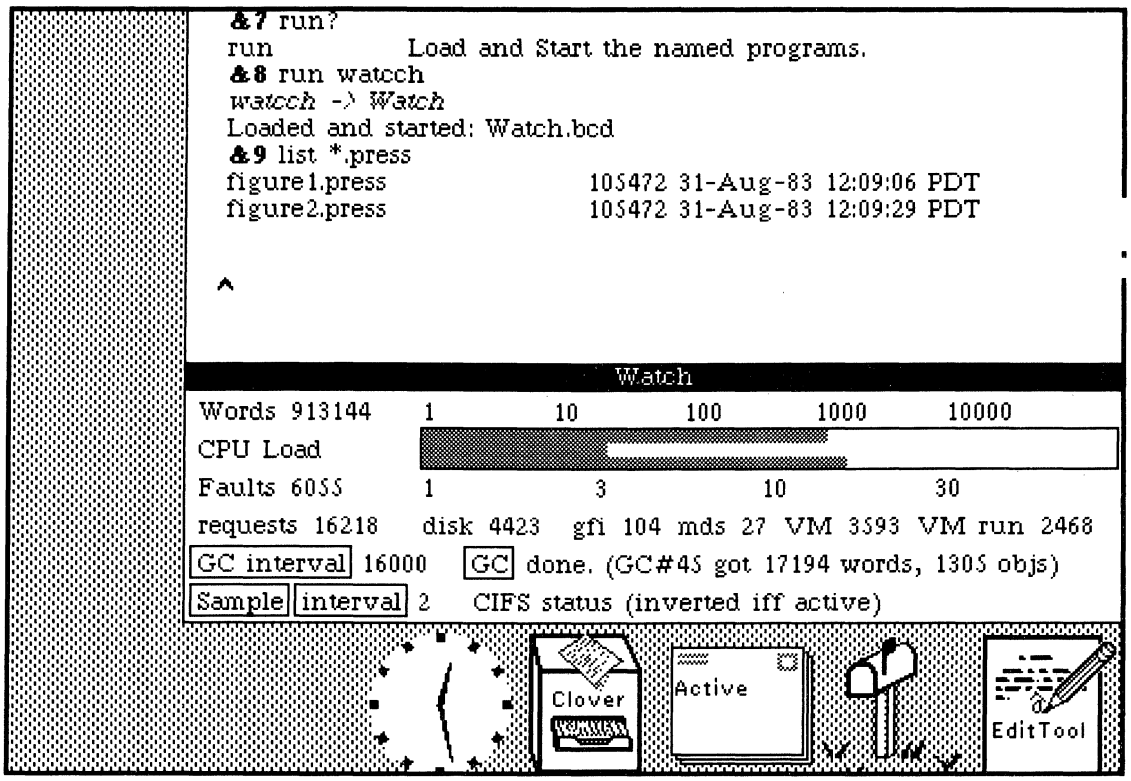


Figure 15

The Watch tool in operation

†51 Cedar programs are often written in expectation of production use of the program. Thus, performance monitoring and tuning is an important issue to Cedar programmers, and "Dynamic Measurement Facilities" was one of the items in our catalogue of programming environment capabilities. The Watch tool is just one example of a number of such tools available in Cedar. Watch is used to give a rough answer to the question "What's it doing" (Watch is also used to answer the question "Is the system still alive?" when an operation seems to be taking an extraordinarily long time.) A much more elaborate and precise performance tool is the Cedar Spy, developed by John Maxwell. "With the Spy, the programmer can see which procedures are consuming CPU cycles, which are causing page faults, which are using the allocator, or which are calling a particular procedure. When the programmer narrows his focus to just one process, the Spy will tell him where that process is spending its time, where it is waiting on page faults, where it is waiting on monitor locks, where it is waiting on condition variables, and when it is preempted by other processes. In addition, the programmer can measure precisely what he is interested in since the Spy provides a facility for setting breakpoints to determine where the Spy should start and stop its measurements" [18].

The Interpreter

One of the valuable lessons we learned from Interlisp and Smalltalk was that the availability of an interpreter greatly facilitates debugging and testing, even when the programs being debugged are themselves totally compiled.†52 Thus, the Cedar interpreter is an important and integral part of the Cedar environment, despite the fact that Cedar is a compiler-oriented language.†53

To show you how the Cedar interpreter works, let's interpret some Cedar expressions. I'll create an interpreter Work Area by clicking the New menu button at the top of Work Area A (Figure 16).†54

---

†52 Actually, all Smalltalk expressions that are input by the user are compiled before execution, although it is not clear that Smalltalk users are (or need to be) aware of this operation. The important point is *the ability to create and execute program fragments in a specified, dynamic context.* Whether this is done via a separate interpreter, as is the case with Interlisp, or by compiling each expression, as Smalltalk does, is simply an implementation issue. We originally had hoped that Cedar could obtain the benefits of an interpreter by appropriately reconfiguring the Cedar compiler. However, the Cedar compiler, having evolved over several years under several implementors, turned out to be so monolithic as to make restructuring it intractable (almost as difficult as starting over from scratch), and so we were forced to implement a separate interpreter.

†53 The availability of a source-language debugger was one of our priority B items in the original EPE report [8]. We felt that: "It is essential that the programmer be able to debug using the same language constructs and concepts used in writing the original program" [8]. To a large extent, we have been successful in meeting this goal with the current Cedar Interpreter. With respect to other uses of the interpreter, the interpreter has not yet reached the stage where it is robust enough, or performs well enough, to allow and encourage the kind of parameterization of programs by expressions that is employed routinely by Lisp programs. The reason for this failure is partly that, for historical and cultural reasons, Mesa programmers simply tend to write applications in a different style than Lisp programmers. However, another reason is that in Cedar, the interpreter was viewed principally as an enabling facility for source-language debugging, and less thought and effort were devoted to the use of the interpreter as a *package* to be invoked by applications programs. Since the availability of an interpreter with low overhead at runtime was rated as only a priority C item, and program-manipulable representation of programs was rated even lower than that, the current state of affairs is not surprising. (There is some debate on this point; some members of the Cedar project do not agree with the author's conclusions.)

†54 Actually there is very little difference between an Executive Work Area and an Interpreter Work Area. Both represent instances of the UserExecutive. The only difference is that the UserExecutive associated with the Interpreter Work Area treats inputs as expressions to be interpreted, whereas the UserExecutive associated with an Executive Work Area expects inputs to correspond to commands to be executed. It is possible (but not considered hygienic) to interpret expressions in an Executive Work Area, and conversely to execute commands in an Interpreter Work Area. In fact, if the user forgets himself and types into an Interpreter Work Area something that does not look like an expression but does look like a command, the system will respond with "Perhaps you meant" followed by the corresponding command, and allow the user to confirm with a single keystroke. (Another example of DWIM.)

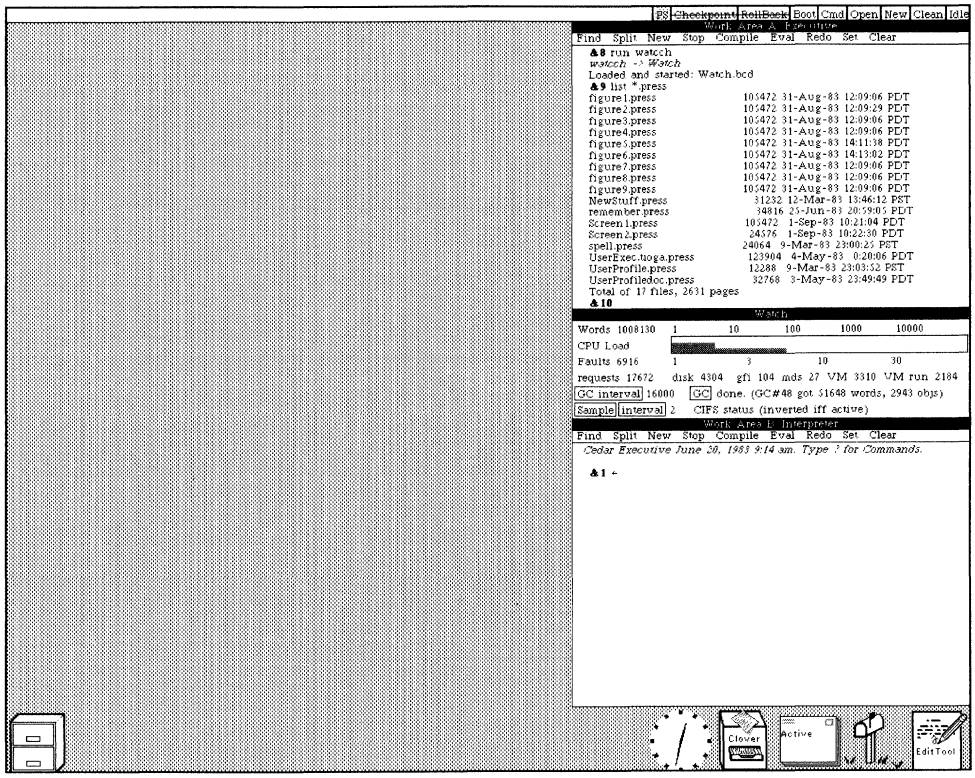


Figure 16

Creating a new interpreter Work Area

The Cedar language includes the data types found in most modern programming languages, such as integers, reals, booleans, characters, arrays, pointers, records, etc.

For example,

```
&1 ← 3 + 4
7
```

a slightly more complicated example:

```
&2 ← ABS[1.414 * 1.414 - 2.0] < .001
TRUE
```

The first event, &1 ← 3 + 4, really means assign the value of 3 + 4 to the variable &1, and I can refer to this value in later expressions.†55 For example, let's multiply it by 1.4:

```
&3 ← &1 * 1.4
9.8
```

†55 All & variables are local to the corresponding executive, i.e., each executive has its own symbol table.

The Cedar interpreter also allows me to perform operations on *types* as well as values. For example, typing ? following an expression will show the type of the value of the expression.†56

```
&4 ← 3.2?
is of type REAL
&5 ← 'X?
is of type CHAR
&6 ← Time.Current?
is of type PROC RETURNS [time: System.GreenwichMeanTime]
```

In the Cedar language, "." is used to denote field extraction. For example, x.y means the field of x whose name is y. In this case, Time is the name of an interface, and Current names a procedure in that interface. An interface is like a *contract* between implementors and clients. It declares that a procedure of a specified name, such as Current, takes certain arguments and returns certain results. The Cedar compiler can then make sure that any programs that *import* (use) this interface conform to its specifications. The compiler also checks that the implementation module conforms to the same specifications.†57

†56 Note that we are not just talking about primitive, built-in data types, such as integer, boolean, string, etc. Cedar encourages the programmer to augment the collection of predefined types by constructing new types defined in terms of built-in or previously constructed types. In a typical Cedar system, there may be over a thousand such types. Thus, for the purposes of debugging, knowing that a particular object is a pointer to a word containing all 0's may not be anywhere near as informative as finding out that the object in question is of type REF Foo, rather than REF Baz, where both Foo and Baz happen to be synonyms for the type INTEGER.

†57 The Cedar Briefing Blurb contains a good introduction to the notions of interface and implementation module: "Although Mesa [and hence Cedar] programs look a lot like PASCAL programs when viewed in the small, Mesa provides and enforces a modularization concept that allows large programs to be built up out of smaller pieces. These smaller pieces are compiled separately, and yet the strong type checking of Mesa is enforced even between different modules. The basic idea is to structure a system by determining certain abstract collections of facilities that some portions of the system will supply to other portions. Such an abstraction is called an *interface* and it is codified for the compiler's benefit in a Mesa source file called an *interface module*. An interface module defines certain types, and specifies a collection of procedures that act on values of those types [e.g., see the Rope interface in Figure 21]. Only the procedure headers go into the interface module, not the procedure bodies. This makes sense, since all the interface module has to do is to give the compiler enough information so that it can type-check programs that use the abstraction. ... The procedure bodies go into a different type of module called an *implementation module*" [23].

The notion of abstraction mechanisms and the explicit notion of interface was a priority A item in our original catalogue of programming environment capabilities: "Abstraction mechanisms are important because they make explicit the logical dependencies of one part of a program on another, while concealing the implementation choices irrelevant to the communication between parts. Thus, these mechanisms enable the ability to factor the development, debugging, testing, documentation, understanding, and maintenance of programs into manageable pieces, while leaving individual programmers the appropriate freedom to design those pieces" [8]. The author believes that the abstract notion of an interface is one of the great strengths of the Mesa programming language. However, the need to specify interfaces in *advance* can also be cited as a weakness of the Mesa approach. Certainly, the present need for vast recompilations whenever a fundamental interface is changed, even in a backwards compatible fashion, is a weakness, but one that certainly can be reduced and maybe even eliminated (for example, by maintaining version stamps at the interface *item* level, rather than at the interface level, as is currently done). It just hasn't happened yet.

Let's call this procedure. It takes no arguments.

```
&7 ← Time.Current[]
Thursday, September 1, 1983 12:33:21 pm
```

Its value is of type:

```
&8 ← &?                                                    †58
is of type System.GreenwichMeanTime
```

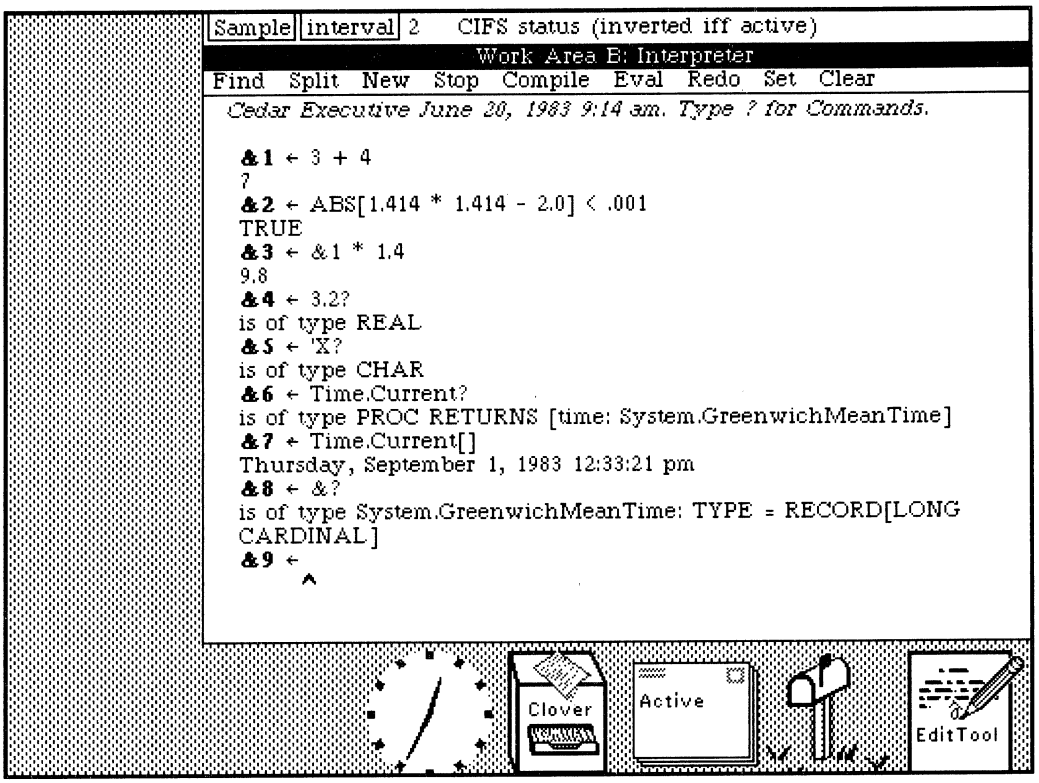


Figure 17

Interpreting expressions

The reason that the value of Time.Current in event number 7 prints so nicely as a day, date, and time, rather than as a 32-bit quantity, is that a *PrintProc* has been associated with the type System.GreenwichMeanTime. A PrintProc is a procedure that provides a more desirable way of presenting an object of a certain type, rather than simply printing its data structure using the default methods. The PrintProc facility is quite useful for dealing with large and complicated data structures such as viewers, documents, and streams, where the user typically just wants to be able to identify the object, rather than seeing its actual structure. Cedar includes a number of PrintProcs for just this purpose. In addition, individual users may define new PrintProcs for their own types. We will see more examples of PrintProcs later.

†58 The value of the variable & is the value of the last event executed, i.e., in this case & and &7 have the same value.

Automatic Storage Management and REFs

In the early stages of planning for Cedar, one of the features that received the highest priority was automatic storage management–a garbage collector.†59 The Cedar language was extended to include a data type called a REF, which is a pointer to an object in collectible storage. In addition to REFs to particular types, such as REF REAL, REF BOOL, REF PROCEDURE, etc., the Cedar language includes a generic REF type, REF ANY.†60

†59 In fact, it was the *highest* priority item, the reason being that it freed programmers from excessive concern for the size and location of their code and data. An excellent account of the importance and effect of the availability of garbage collection on programming style in Cedar is contained in [23]: "The programming language underlying Cedar is essentially Mesa with garbage collection added. Adding garbage collection actually changes things quite a bit. First of all, it changes programming style in large systems tremendously. Without garbage collection, you have to enforce some set of conventions about who owns the storage. When I call you and pass you a string argument, we must agree whether I am just letting you look at my string, or I am actually turning over ownership of the string to you. If we don't see eye to eye on this point, either we will end up both owning the string (and you will aggravate me by changing *my* string!) or else neither of us will own it (and its storage will never be reclaimed–a storage leak). Once garbage collection is available, most of these problems go away: God, in the person of the garbage collector, owns all of the storage; it gets reclaimed when it is no longer needed, and not before. But there is a price to be paid for this convenience. The garbage collector takes time to do its work. In addition, all programmers must follow certain rules about using pointers so as not to confuse the garbage collector about what is garbage and what is not."

It is only fair to observe that the above statement "it gets reclaimed when it is no longer needed" is not strictly true: circular, or self-referencing structures are only reclaimed by the trace-and-sweep garbage collector, which must be explicitly invoked. However, data structures that are not self-referencing are automatically reclaimed by the incremental garbage collector, which runs all the time as a background task.

†60 A recurring theme in our discussions of requirements for an experimental programming environment centered around the issue of early versus late binding of various implementation decisions. On this subject, Beau Sheil [27] observed that: "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. ... The languages make extensive use of late binding, i.e., allowing the programmer to defer commitments as long as possible" [27].

The addition of the type REF ANY to the Cedar-Mesa language represents an attempt to provide for one form of late binding; use of the type REF ANY enables an implementor to defer type checking from compile time to runtime on a case by case basis. Note that in the Lisp programming language, every item is effectively a REF ANY: all objects are pointers, and the type of each object can always be determined at runtime. As a result, certain classes of errors can remain undetected until a program is run, perhaps even until the program is run on particular data. At the other extreme, the Mesa programming language requires the specification of the type of each object at compile time. Consequently, unanticipated modifications or extensions to Mesa programs often require changes to type declarations and recompilation of interfaces and implementation modules.

In Cedar, we wanted the best of both worlds: the flexibility of runtime (dynamic) type checking and the reliability and performance of compile-time (static) type checking. We hoped that by employing REF ANY in the early stages of development, programs could opt for more flexibility at the expense of performance and/or runtime errors. As the program matured, various binding decisions could be made earlier by employing specific types where appropriate.

Another important use of REF ANY in Cedar is to enable generic programs. Since programs can determine the type of a REF ANY at runtime, they can operate differently depending on the type of the object they are given. For example, the same Sort program can be used to sort lists of integers, reals, strings, or even viewers, by selecting the appropriate comparison algorithm based on the type of the objects being compared. The capability provided by REF ANY is also essential for enabling object-oriented programming. For example, streams, viewers, and ropes are all objects in Cedar whose definition consists of a block of procedures along with a datum which contains the state of the object. Since the type of the datum is different for each different implementation, for example, file streams need different information than keyboard streams, the datum is represented as a REF ANY which the individual procedures can then interpret.

Atoms, which are very similar to Lisp atoms,†61 and Lists are also examples of REFs.†62

For example, let's make a list of some of the values that we just computed.

```
&9 ← LIST[&1, &2, &3, &7]
(↑7, ↑TRUE, ↑9.8, ↑Thursday,  September 1, 1983 12:33:21 pm)†63
&10 ← &?
is of type LIST OF REF ANY
```

Since each of these objects is of a different type, the type of the value of event 9 is LIST OF REF ANY. Note that the first element is really a REF INT, the second a REF BOOL, the third a REF REAL, etc. In other words, the type of &9.first, the first element of this list, is REF ANY, but the type of the referent of this element, &9.first↑, is INT.

---

†61 In order to ascertain the degree and nature to which atoms were actually used in Cedar, the author undertook an informal canvass of Cedar users. This footnote reports on the results of that poll.

One of the principal uses made of atoms is to provide for a form of late binding: provide the client with an open-ended enumeration at run-time (with correspondingly less compile-time checking), as opposed to the standard Cedar enumerated type in which each element of the enumeration must be specified at compile-time. For example, one user reported: "In some situations, I pass a general atom instead of an element of a specific enumeration in order to avoid recompilation when I add a new element (especially to an error enumeration when the list of errors is not quite clear). Later I convert to a specific enumeration to gain the tighter binding."

Another use made of atoms in Cedar takes advantage of their unique print names, i.e., there is a one-to-one mapping between a sequence of characters and an atom (a 32-bit quantity), and it is very cheap to compare atoms for equality (which is not the case for comparison of two sequences of characters). Applications take advantage of this fact to save space and time

A third use made of atoms in Cedar involves property lists. Each atom has a property list associated with it; applications use these property lists to provide for unforeseen extensions. However, some applications such as Viewers and Tioga prefer to include a separate property list as part of the data structure for the corresponding object, rather than using the more global atom property list. "Property lists attached to objects are wonderful, but I think 'global' property lists attached to the atoms themselves are probably a bad idea." Associating property lists with the objects themselves also provides a place for clients to store information associated with the object that the client can then subsequently interpret and use.

†62 A List in Cedar is a REF to a structure consisting of two fields, first and rest. The first field contains the element of the list and the rest field the tail of the list (the Lisp CAR and CDR). Cedar provides language support for the construction of lists (via LIST and CONS), but no polymorphism; it is not possible to write a program that traffics in LIST OF T without specifying T at compile time. Since most programs using lists employ lists of specific types, the absence of polymorphism means programmers must (re)implement for each specific type list primitives such as Reverse, Append, Union, and Intersection. This absence of polymorphism is cited as the biggest shortcoming of the current implementation.

†63 ↑ is how Cedar prints REFs, e.g., ↑7 is a REF to the object that prints as 7.

Manipulating Lists

The List interface includes a variety of procedures for manipulating lists. Let's create a viewer on the List interface and look at it. I click the **New** button in the message area at the top of the screen to create a new viewer on the left. Then I type the name of the interface into this viewer to cause the corresponding file to be loaded into the viewer, as has been done in Figure 18.†64

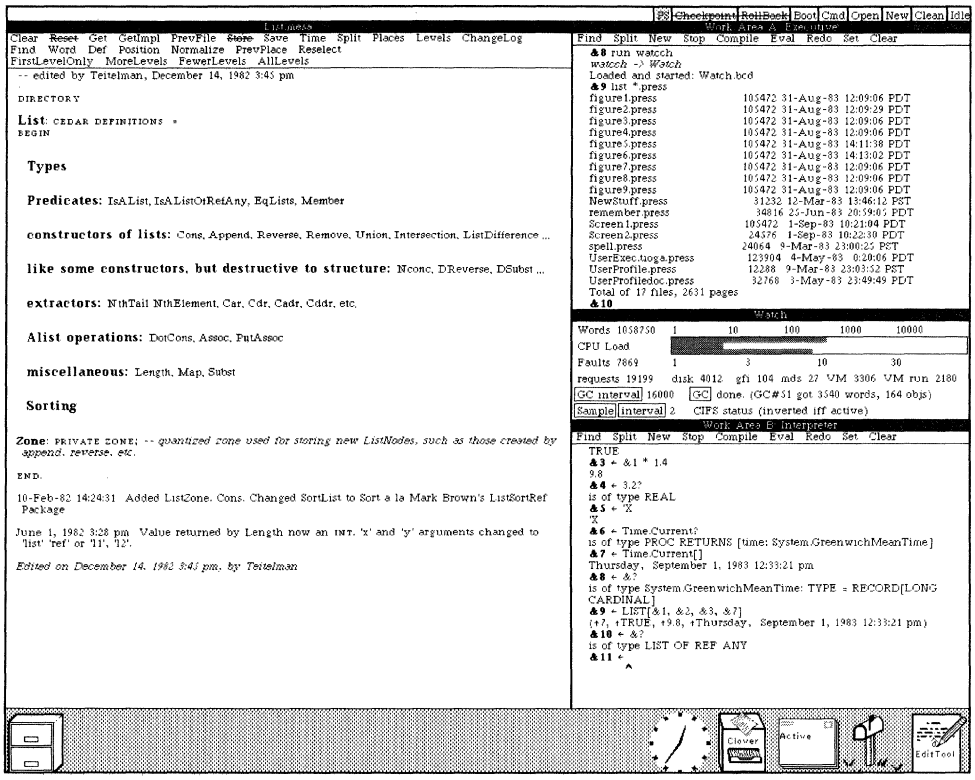


Figure 18

The List interface contains procedures for creating and manipulating lists

Let's try the procedure **Reverse** on the list that we constructed in event 9. We return to our interpreter Work Area on the right and type...

&11 ← List.Revers[&]
Revers -> Reverse ?

†64 As mentioned earlier, we also use the Tioga editor for creating and modifying Cedar programs. The List interface shown in Figure 18 is in fact a Tioga document. Note that the use of node structure and levels here effectively provide a table of contents. The same node structure also enables the user to manipulate program statements, blocks, etc., as single entities, even though Tioga does not know about the Cedar language syntax.

I misspelled the name of the procedure causing an error to occur, i.e., the procedure Revers was not found in the set of procedures contained in the interface List. DWIM was invoked and searched through the set of items declared in the interface List.†65 DWIM found a procedure, Reverse, whose spelling was pretty close to what I typed, and in Figure 19, DWIM is now waiting for me to confirm or reject the correction,†66 which I can do via the keyboard, or by clicking the Yes or No menu buttons which have been added to the Work Area's menu for this purpose (just above the arrow-shaped cursor in Figure 19).†67 When (and if) I confirm the correction, the corrected expression will be evaluated.

---

†65 When we first began work on Cedar; some thought that the complexity of the Cedar language would make it too difficult to implement any sort of automatic error-correction facility such as was available in Interlisp. However, this very complexity turns out to be of great benefit for error correction in Cedar expressions, because more information is available at the time of the error than with Lisp, where all the interpreter knows is that an identifier is unrecognized and whether it was used as a function or a variable. For example, when the user typed List.Revers above, DWIM was called given the identifier "Revers," the message "selection failed," and the context the List interface. DWIM knew that it was looking for an element defined in the List interface, which immediately narrowed the search down to 42 possible candidates. Similarly, List.Subst is a procedure which takes three arguments whose names are new, old, and expr. If the user types List.Subst[new: $Foo, old: $Fie, exrp: x] (misspelling the name of the third argument), then DWIM only has to consider three candidates. For assignments, the type of the target can also be used to guide the correction. For example, if x is declared to be of type Color, where Color is an enumerated type consisting of {red, green, blue}, and the user writes x ← bue, then he probably means blue, whereas if x is of type {feature, nonfeature, bug}, and the user writes x ← bue, he probably means bug.

†66 The algorithm for spelling correction and confirmation is the same as that used in Interlisp [14]. Basically, a metric is computed which measures the distance between two tokens in terms of the number of characters that do not match. If all characters are accounted for, i.e., the only errors are transpositions or doubled characters, then confirmation is not required. In the case shown here, a character was missing, so confirmation was required. However, the user can specify in his user profile a default timeout and value for confirmation. In this case, when confirmation is required, if the user does not respond within the indicated interval, the value specified as a default is taken as the response. For example, if my default timeout is 60 seconds and the default value is Yes, then if I type ahead a sequence of operations and go to lunch, the system will wait 60 seconds for me to confirm a correction, and then proceed with the correction. On the other hand, a conservative user may not want the system to make any corrections without confirmation. User profiles in Cedar allow users to customize the behavior of the system to suit their own preferences.

†67 In general, we try to give the user the choice of performing operations either via menu or via the keyboard. The main reason for this redundancy is that if the user's hands happen to be on the keyboard, it is more convenient to interact through that medium rather than having to reach for the mouse. Conversely, if the user's hands are already on the mouse, it is easier to click a menu button than to reach back to the keyboard. The use of menus in conjunction with confirmation provides the added benefit of allowing the system to gracefully handle the issue of type-ahead and its potential interaction with confirmation. Consider the case where the user has entered some operation, and then typed ahead the next operation not realizing that the first would require some kind of confirmation. The desired behavior of the system is that the user be able to confirm without having his type-ahead affected, i.e., that he not have to retype it after confirmation. This is accomplished by requiring that the user only confirm via menu once there has been any type-ahead.

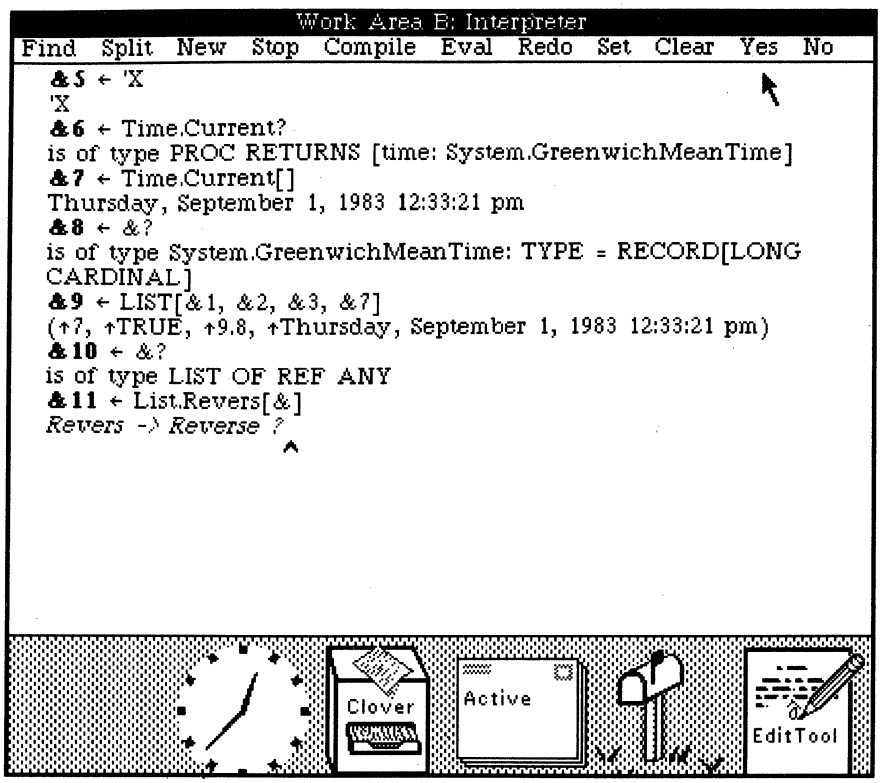


Figure 19

Confirming a DWIM error correction

&11 ← List.Revers[&]
Revers -> Reverse ? Yes
(↑Thursday, September 1, 1983 12:33:21 pm, ↑9.8, ↑TRUE, ↑7)

Ropes

Cedar also includes another useful type of REF called a ROPE. A ROPE is Cedar's standard string type.[†68] The input syntax for a ROPE is a sequence of characters delimited by "'s. For example:

```
&12 ← "this is a rope"
"this is a rope"
&13 ← &?
is of type ROPE
```

Just as the List interface provides operations for dealing with lists, the Rope interface contains a variety of useful operations on ROPEs. For example, Rope.Find is a procedure that searches one ROPE for the occurrence of another.

```
&14 ← Rope.Find?
is of type PROC [s1: ROPE, s2: ROPE, pos1: INT ← 0, case: BOOL ← TRUE] RETURNS [INT]
```

This tells us both the names and the types of the arguments that Rope.Find expects, and that it returns an integer. (This integer indicates the character position in the first ROPE at which the second ROPE begins.) Let's try it.

```
&15 ← Rope.Find[
```

At this point, instead of retyping the ROPE "this is a rope," I can simply select the corresponding text in event number 12 using the mouse, and cause the characters to be treated exactly as though they had been typed. I can do this because this Work Area I have been typing to as though it were simply a glass teletype is really a full-fledged Tioga document, and I can make use of any of the facilities of the Tioga Editor when constructing expressions to be interpreted. For example, if I hold down the SHIFT key while selecting in a Tioga document, the selected material is displayed with a gray underline (as is shown in Figure 20). Such a selection is called a *source* selection. When I release the SHIFT key, this source selection will be copied to the current insertion point, i.e., the place where the caret is.[†69]

---

†68 A ROPE is a garbage-collectible sequence of characters. ROPEs are immutable; the sequence of characters denoted by a ROPE never changes. Thus, ROPEs may be shared freely among independently-written applications, since no application can hand out a ROPE and have some client free its storage or somehow alter the characters it contains. ROPEs are also more general than conventional strings: a client can provide his own specialized implementation of a ROPE by implementing a small set of basic operations on the new representation, and applications that traffic in ROPEs need not distinguish between these specialized ropes and the standard variety. In other words, the Rope interface treats a ROPE as an object (in the Smalltalk sense) which knows how to perform certain operations. (Such object-style programming is generally recognized as a good thing, but except for some isolated instances, has not caught on in general with the Cedar community. Some of this is due to identified and understood language deficiencies.) Ropes were designed and implemented by Russ Atkinson. It is generally agreed that ROPEs are something that Cedar got right.

†69 This feature is tremendously useful. It greatly increases the bandwidth of the user's interaction with the system. It also enables the use of long and descriptive identifiers, such as IO.CreateEditedStream, UserExec.FindExecFromViewer, and ViewerTools.GetSelectionContents, even though many of our users are not fast typists. Such long identifiers are tolerable because they *rarely have to be typed*, but usually can be copied from somewhere else on the screen, e.g., from a viewer on the interface that defines them. (Note that having to *read* long identifiers in programs is *not* a burden, but in fact is an asset, since the name contains so much information it is, in effect, a form of documentation.)

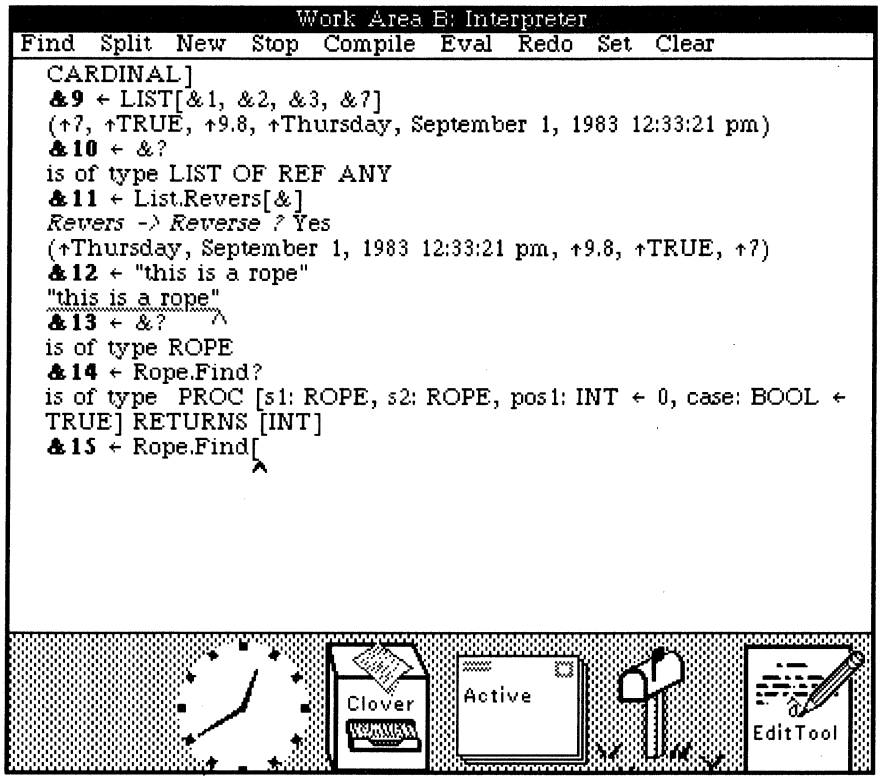


Figure 20

Source selections permit copying characters from one place to another as an alternative to typing

```
&15 ← Rope.Find["this is a rope", "is a"]
5
```

The value 5 indicates that the second ROPE begins at character position 5 in the first ROPE.

This gives you a general overview of the Cedar interpreter. Now let's try using the Cedar system in earnest.

Tracking Down a Bug

Earlier when I typed Rope.Find? in event 14, the system simply told me the names and types of the arguments and return values. I thought that the system was also supposed to show me the comments associated with the procedure Find in the Rope interface, so that I would know what the various arguments and the return value meant. Let's open this interface and see if there are any comments associated with this procedure. Rather than creating a new viewer, I'll simply reuse the viewer on the List interface. I select Rope.Find in my Work Area, and then click the Get menu button in the viewer. This tells the Viewers Package to load the file Rope.mesa into this viewer, and then search for the definition of the procedure Find, which it has finished doing in Figure 21.

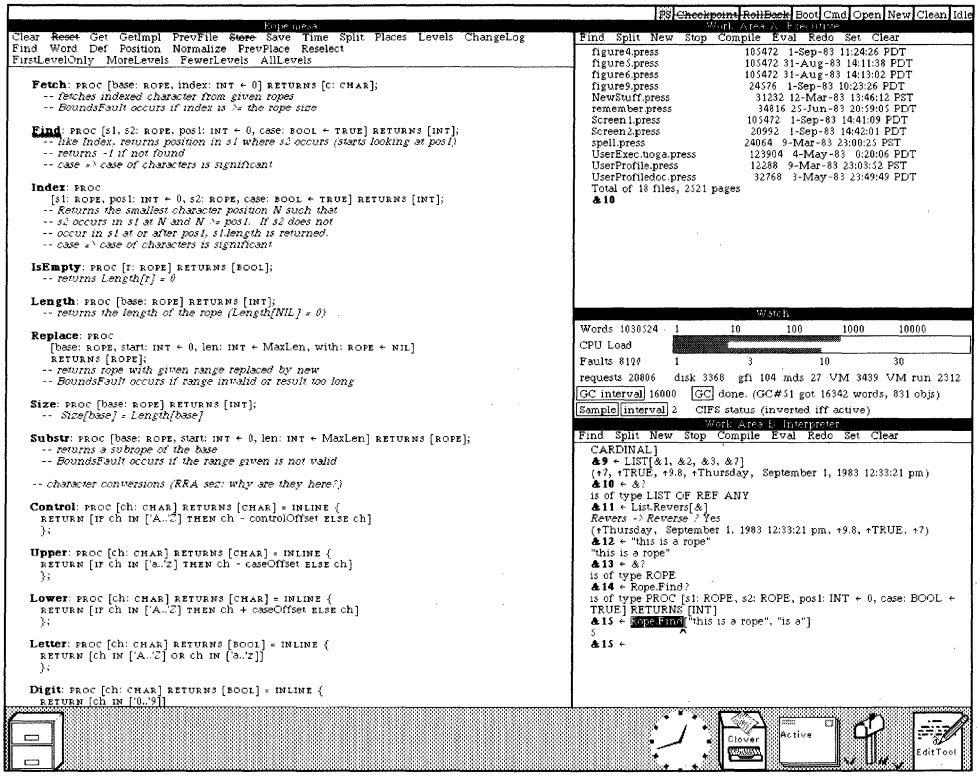


Figure 21

Convenient access to program sources provides a form of online documentation

As you can see, there are comments here. Let's try to find out why they weren't shown when I typed "?". To do this, I am going to plant a breakpoint in the code that implements the ? feature of the UserExecutive. First, I create a viewer on the corresponding source file, as shown in Figure 22.

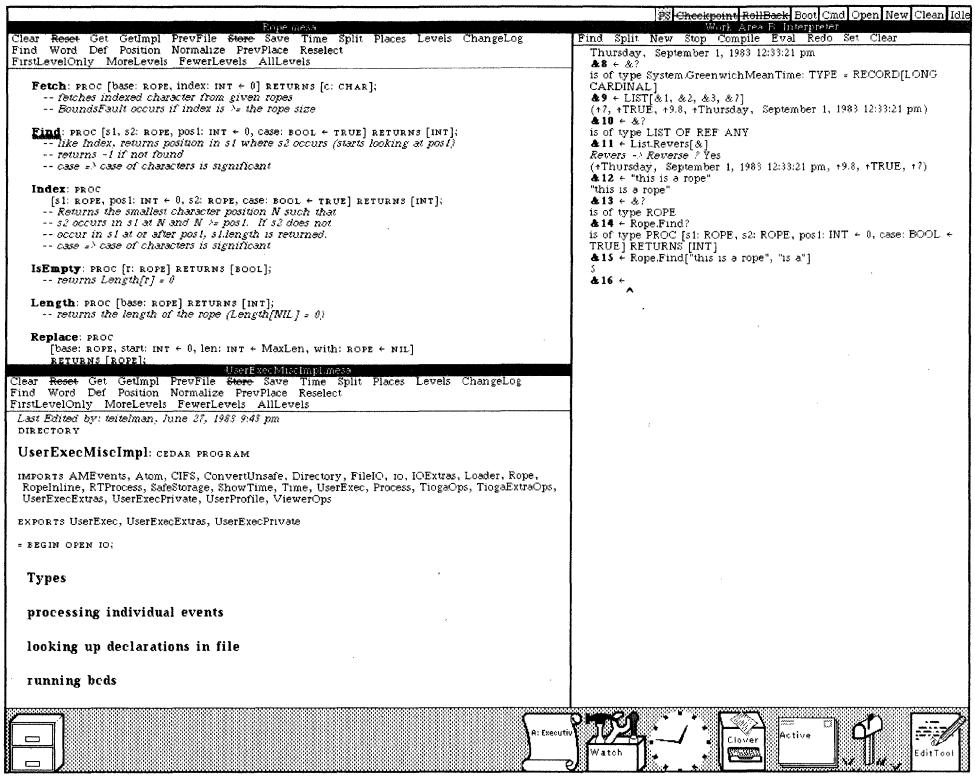


Figure 22

Opening a viewer on a source file in preparation for setting a breakpoint

I then scroll to the section entitled "looking up declarations in file," click the MoreLevels menu button a few times, and select a location within the procedure PrintDeclFromSource where I want the breakpoint inserted.†70 I then plant the breakpoint at this location by clicking the Set menu button in my Work Area.†71

```
&16 SetBreak UserExecMiscImpl.mesa 13897                    Break #1 set.
Break #1 in UserExecMiscImpl.PrintDeclFromSource (source: 13891)
    pattern ← TiogaOps.CreateSimplePattern[target]; -- creates a pattern for the search.
```

†70 The reader may wonder how I knew where to place the breakpoint. In this particular case, I happened to be familiar with the internal workings of the UserExecutive. However, it is not at all uncommon for Cedar users, especially experienced ones, to poke around in other people's code, planting breakpoints, examining data, etc. This behavior is facilitated by the use of long, suggestive names as well as the structuring of the source files that Tioga enables. As a result, it is not uncommon for a bug report not only to describe the symptom, but to identify the offending line of code.

†71 Clicking the Set menu button causes an appropriate command line to be constructed and input to the UserExecutive, rather than executing the operation directly. This technique provides the user with a record of all of his interactions with the executive and enables him to examine or replay them at some later point.

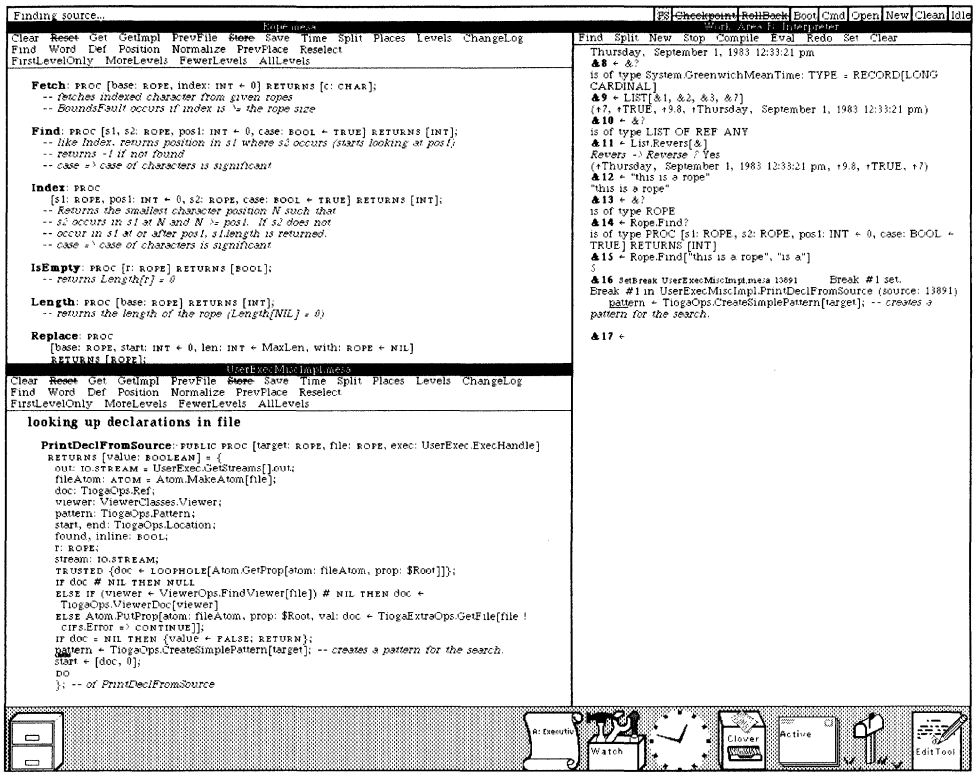


Figure 23

Setting a breakpoint

The breakpoint has been set, as shown in Figure 23.†72 The system provides feedback by displaying in my Work Area the corresponding line of source text with the location of the breakpoint underlined, as well as by underlining the corresponding location in the source viewer (in Figure 23, the bottom viewer in the left column).

Now let's reexecute Rope.Find?. I can do this by simply selecting anywhere inside of the corresponding event and clicking the Redo menu button. The UserExecutive maintains a history of the events that have been executed.†73 It uses this history list to find the event corresponding to my selection and reexecute it.†74

†72 Setting a breakpoint involves finding the place in the object (compiled) code that corresponds to the indicated location in the source, and then inserting a special instruction that will invoke the breakpoint machinery. The Cedar compiler facilitates this process by constructing as a by-product of compilation a table that contains for each statement the mapping from the object locations to the corresponding source location. However, most users are unaware of this process, and simply think of and treat the source file as *the* program. Cedar goes to great lengths to encourage this model.

†73 The notion of a history list and facilities for manipulating it came from Interlisp. We have not yet implemented the notion of Undo as applied to events that Interlisp provides. This is partly because it is harder to capture all of the side effects of an operation in a language such as Cedar, and partly because other tasks were given higher priority.

†74 The user can also reexecute events by selecting the characters that were originally typed while holding the SHIFT key down, as was done in event 15. The principal convenience of the REDO menu button is (a) the user can simply select anywhere in the event, and (b) multiple events can be reexecuted by selecting a range that spans the desired events.

&17 Redo 14
>← Rope.Find?
is of type
Break #1 in UserExecMiscImpl.PrintDeclFromSource
computation suspended, switching to Action Area C...

(and down below a new Work Area pops up in which appears:)

Action #1 (kind: break, process: 173B) (from Work Area B)
Break #1 in UserExecMiscImpl.PrintDeclFromSource
pattern ← TiogaOps.CreateSimplePattern[target]; *-- creates a pattern for the search.*

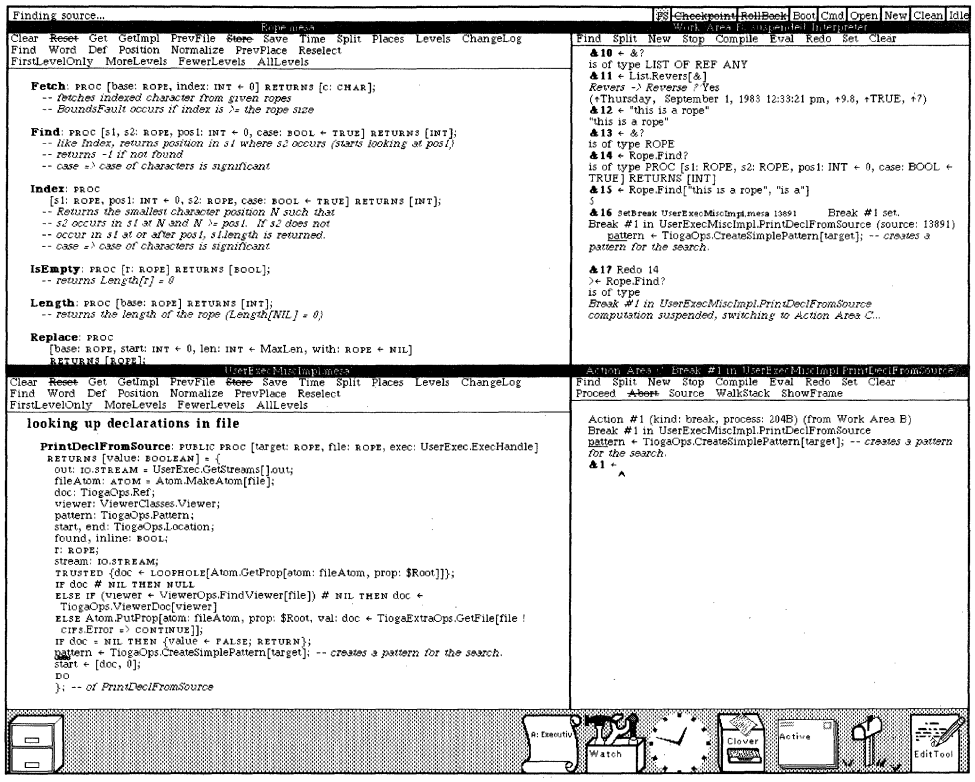


Figure 24

Hitting a breakpoint

Breakpoints and Action Areas

Whenever a breakpoint is encountered in Cedar, the corresponding process is suspended so that the user can examine the state of the computation. We have found it useful for these interactions to take place in an entirely separate Work Area called an Action Area.†75 In Figure 24 we see that a new Action Area has been created. This Action Area tells me that I am at a breakpoint that arose out of an operation in Work Area B. It also tells me that the breakpoint is in the procedure PrintDeclFromSource, and shows me the line of code in which the breakpoint occurred.

The first thing I want to do in this breakpoint is to examine the arguments to the procedure PrintDeclFromSource. To do this, I middle-click the ShowFrame menu button in my Action Area.†76

```
&1 ShowFrame args    UserExecMiscImpl.PrintDeclFromSource
A- target: "Find\n",                          †77
   file: "Rope.mesa",
   exec: {UserExecHandle: "B"}                †78
```

The debugger tells me that this procedure, PrintDeclFromSource, has three arguments, target, file, and exec. The values for file and exec are ok, but the value of target should be "Find" rather than "Find\n." Let's see if this is the only problem, i.e., if target were "Find," would the comments be printed? So I reset the variable target using the interpreter.

```
&2 ← target ← "Find"
"Find"
```

Now I'll allow the computation to continue by clicking the Proceed menu button, and we'll see if the comments from the Rope interface are in fact printed in Work Area B above.

---

†75 This method also supports the Principle of Non-Preemption espoused in footnote 46. The user is not required to deal with this action at this time. He can continue editing documents, create and interact with other executives, read his mail, etc., and this action will wait for him. Another benefit of separate Action Areas is that it enables the user to keep track of the flow of control if another action occurs while pursuing this one.

†76 Left-clicking this menu button would show me just the frame's name, right-clicking would show the name, arguments, plus the local variables. There is some controversy over this overloading of menu buttons, especially when the use of the various mouse buttons is further inflected via the CTRL or SHIFT keys. On the one hand, some users feel that it makes the interface too complicated to learn. On the other hand, there is the desire on the part of experts who are facile with the system to be able to perform complicated operations with a minimum of keystrokes and mouse actions, and the competition for screen real estate (there is room for only so many buttons). Our current plan is to try to satisfy both camps by providing for pop-up menus that will allow the novice (or forgetful) user to peruse all of his options, while retaining the ability for allowing the expert to specify his own abbreviations via various mouse and CTRL/SHIFT combinations.

†77 \n is how Cedar prints carriage-return when it appears as part of a value.

†78 The printing of UserExec handles is another example of the use of PrintProcs. The actual handle is a fairly complicated data structure.

&3 Proceed
proceeded Action #1, returning to Work Area B
~~~~~~

(and in Work Area B above:)

```
PROC [s1, s2: ROPE, pos1: INT ← 0, case: BOOL ← TRUE] RETURNS [INT];
   -- like Index, returns position in s1 where s2 occurs (starts looking at pos1)
   -- returns -1 if not found
   -- case  = > case of characters is significant
```

and sure enough, there are the comments.

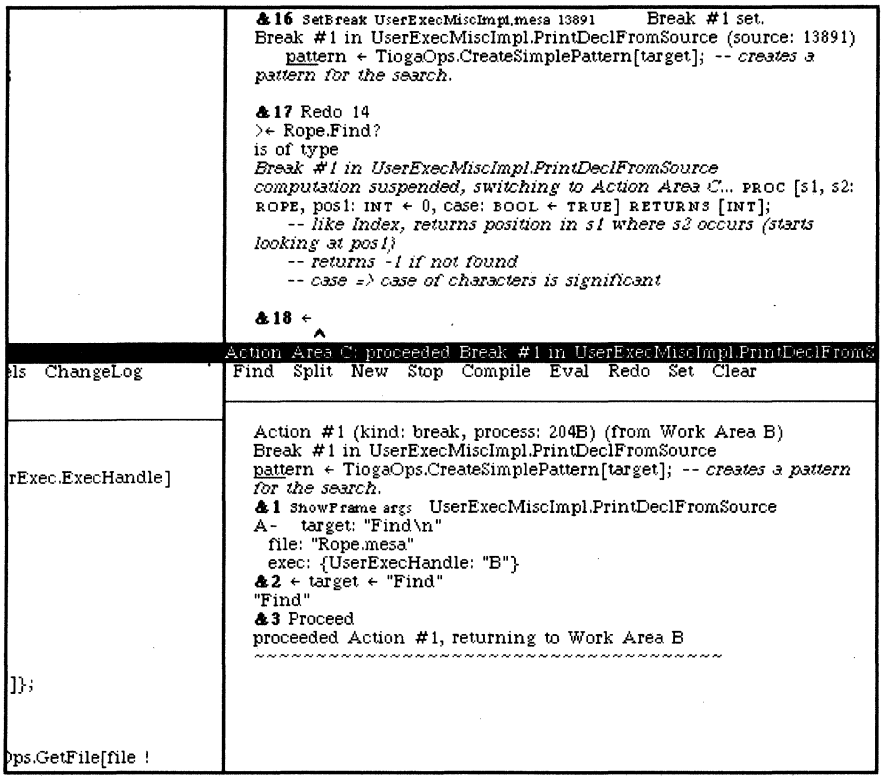


Figure 25

Testing a proposed bug fix by manually resetting data and proceeding

Having identified the nature of the problem, now we must find out the cause–why is the wrong value being given for target in the first place? Let's redo Rope.Find? again...

&18 Redo 17
>← Rope.Find?
is of type
Break #1 in UserExecMiscImpl.PrintDeclFromSource
computation suspended, switching to Action Area C...
~~~~~~

... and we are back at the breakpoint. Now I'll use the WalkStack menu button to climb the call stack. Each time we click the WalkStack menu button, we climb/descend the call stack one frame.†79

&4 WalkStack UserExecMethodsImpl.Help

Now we are at the frame corresponding to the procedure that called PrintDeclFromSource. I'd like to look at the source code corresponding to this call. I click the Source menu button, and the system will find the source and display it in a new viewer on the left.†80

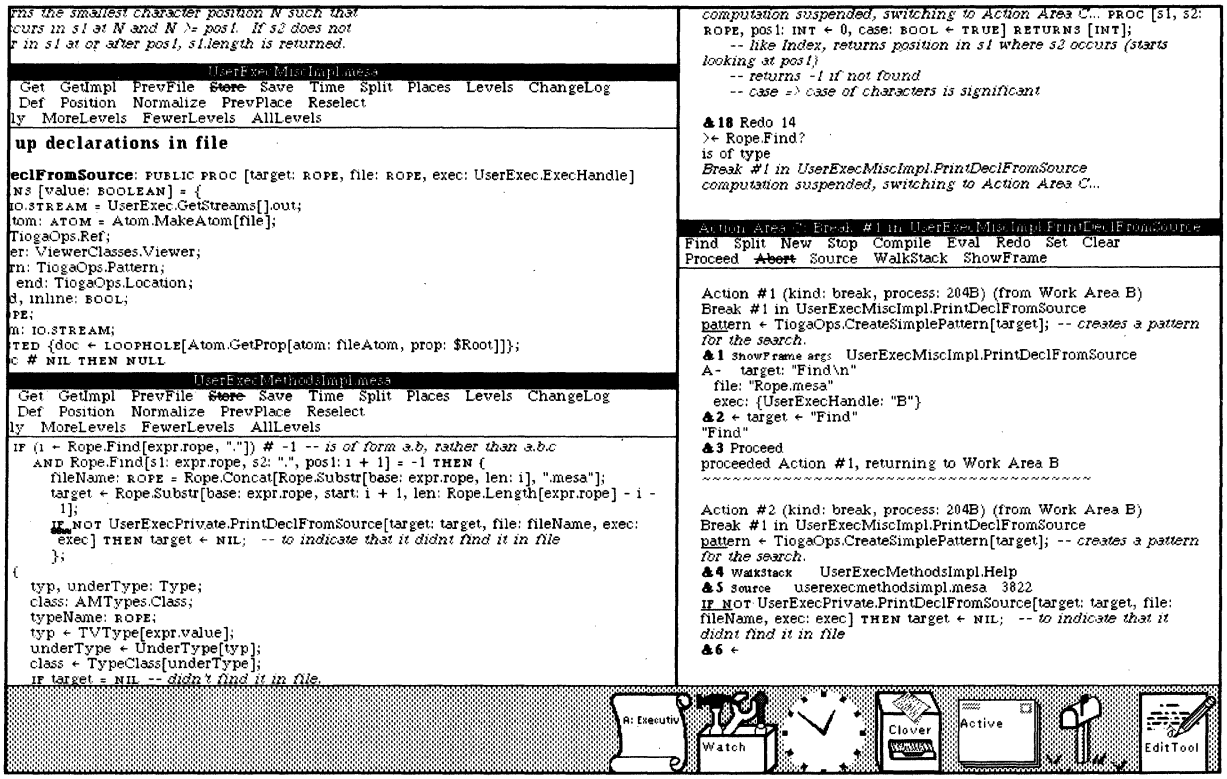


Figure 26

The Source command finds the location in the source file corresponding to the call stack

&5 Source userexecmethodsimpl.mesa 3822
IF NOT UserExecPrivate.PrintDeclFromSource[target: target, file: fileName, exec: exec] THEN target ← NIL; -- *to indicate that it didnt find it in file*

The underlined location is the point in the procedure UserExecMethodsImpl.Help that corresponds to where the computation is right now, i.e., the statement from which PrintDeclFromSource was called. Notice in Figure 26 that immediately before this statement is the expression: target ← Rope.Substr[base:

†79 Left-clicking climbs, right-clicking descends.

†80 This operation involves using the compiler's statement map to perform the inverse mapping from that of planting breakpoints, namely given a location in object code, find the corresponding location in the source. If the source file is not on the user's local disk, but is part of the released system, i.e., is contained in the version map (see footnote 28), the file will be automatically obtained from a file server.

expr.rope, start: i + 1, len: Rope.Length[expr.rope] - i - 1]. This expression uses the procedure Rope.Substr to compute target as the substring of expr.rope that is len characters long, and begins at position start. We already determined in the previous breakpoint that the value of target at this point is the ROPE "Find\n," instead of the ROPE "Find." Let's find out why this is the case by examining the arguments specified in the call to Rope.Substr. First, we'll find out the value of the argument named base by evaluating the expression expr.rope. We can do this by simply pointing at the expression in the source viewer while holding down the SHIFT key, thereby causing the characters to be copied into the interpreter Work Area, the same as we did earlier, even though in this case we are copying characters from one viewer into another.

```
&6 ← expr.rope
"Rope.Find\n"
```

That's what we expected. Similarly, let's check the value of start, the starting position for the substring, specified to be i + 1, and the value of len, the length of the substring, given by Rope.Length[expr.rope] - i - 1. Figure 27 shows the display as I am about to evaluate this latter expression. (Note the source selection underlined in gray in the viewer on the lower left.)

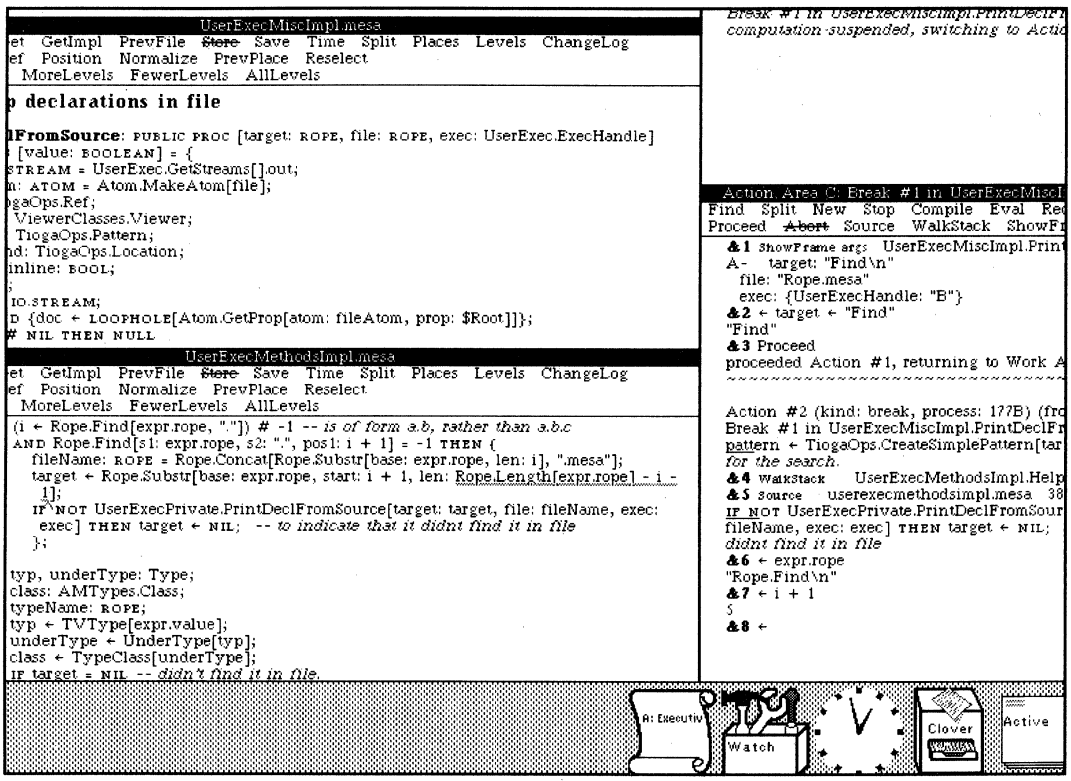


Figure 27

Evaluating expressions from a source program by pointing at them

```
&7 ← i + 1
5
&8 ← Rope.Length[expr.rope] - i - 1
5
```

Here is the problem, an off-by-one bug. If we don't want the \n to be included, there should only

be four characters in the substring, instead of five. In other words, the length argument should be the length of the entire ROPE, minus the start position, minus 1 (so as not to include the last character), i.e., Rope.Length[expr.rope] - (i + 1) - 1. Let's make that change in the source.

I make the edit using Tioga, and then click the ChangeLog menu button. This automatically constructs a change log entry containing my name, the date, and a list of those items that have been changed. It also provides a space for me to fill in a comment describing each change, as shown in Figure 28.

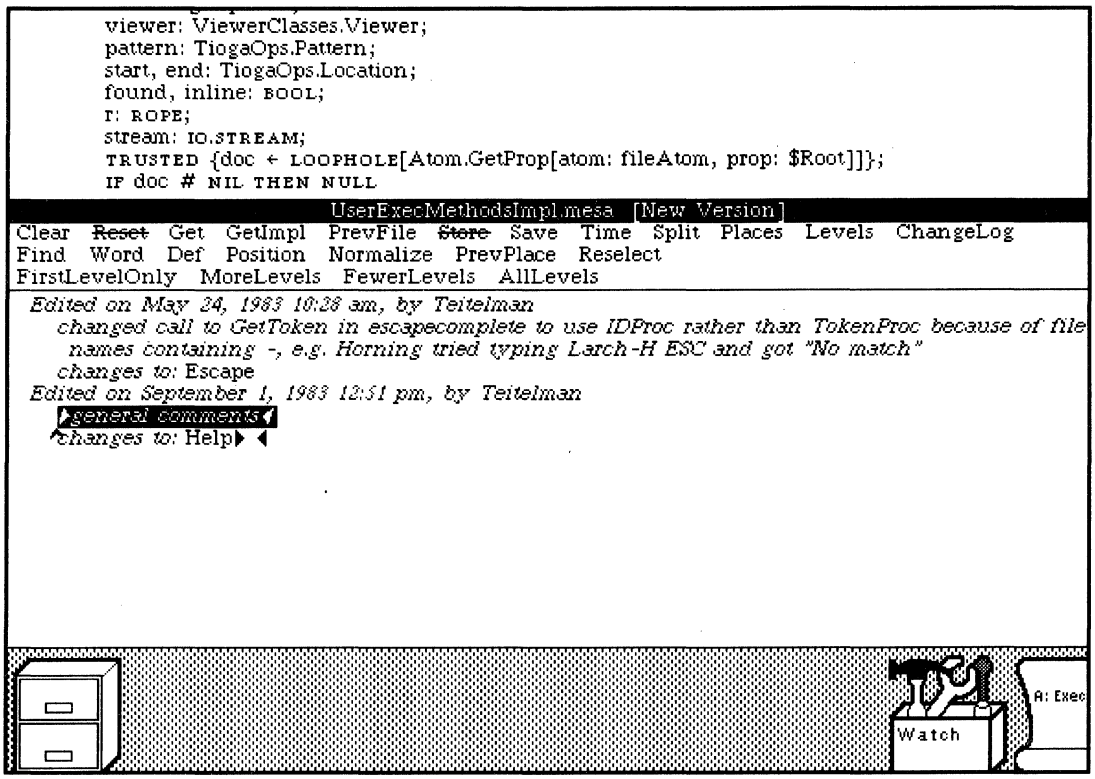


Figure 28

Automatic ChangeLog maintenance

I fill in the comments field, and then save the file using the Save menu button.†81

Now let's go back to the Action Area on the right, clear the breakpoint, and since we are finished with this problem, let's just abort the action and control will return to the Work Area from which the action originated.

†81 Had I just clicked the Save menu button without clicking the ChangeLog button first, the system would have automatically constructed a ChangeLog entry containing my name, the date, and a list of items that had been changed, but without any explanation of the reason behind or nature of the change. However, even this amount of information can be extremely useful in an environment in which several different programmers may edit the same program.

Electronic Mail

The next thing I want to do is to fix a bug that was reported to me in a message. As I mentioned earlier, the mail box shaped icon in the lower right corner of the screen (see Figure 1) is my Walnut Control Panel. I'll open it now. As the flag on the mail box icon indicated, the Walnut Control Panel tells me that I have new mail. I click the NewMail menu button in the Walnut Control Panel to retrieve these messages from the mail server. These messages will initially be placed in my Active message set, represented by the icon that looks like a stack of envelopes. I'll open my Active messages and we will be able to see my new mail.†82

The messages marked with ? in Figure 29 are the ones that I haven't read yet.†83 Some I'll simply read and delete (or delete without reading because the subject does not interest me, e.g., "eye glasses

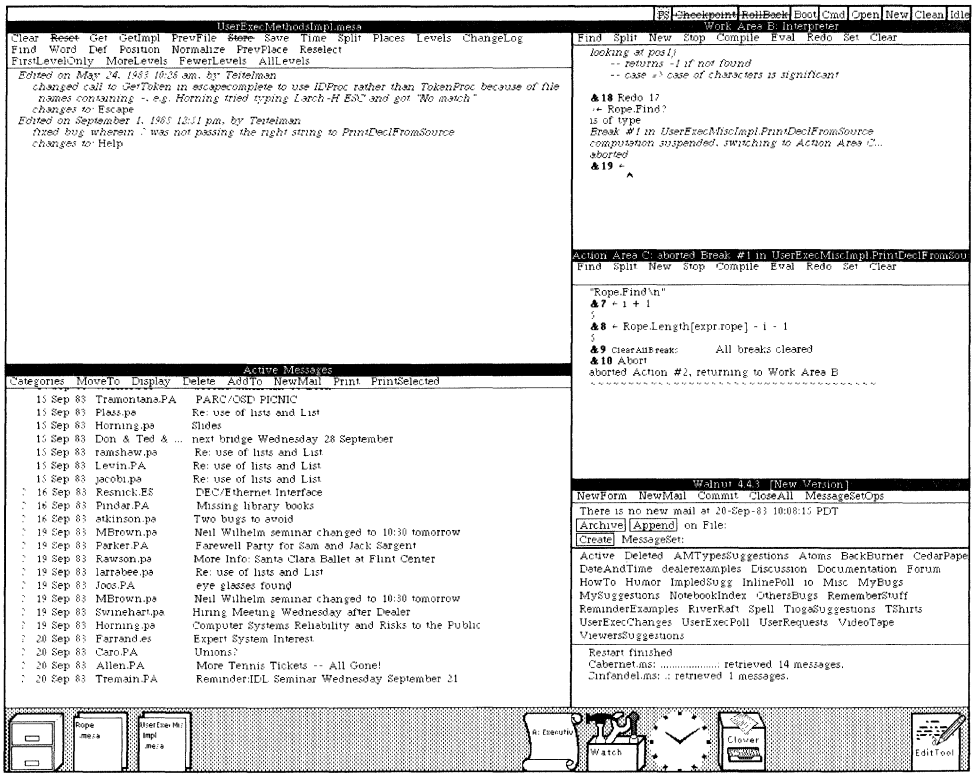


Figure 29

My Active message set and the Walnut control panel

---

†82 We rely heavily on our electronic mail systems at PARC. We use them for mail as well as for the type of announcement that might in other environments be posted on a bulletin board. In addition to messages from one user to another, announcements of impending meetings, for sale notices, and the like are all sent as messages directed at expansive distribution lists. You can see examples of such messages in my Active Message Set in Figure 29 (bottom viewer, left hand column).

There are a number of such electronic mail systems in use at PARC (because there are several different programming environments). However, all of these access a common mail distribution service [1]. Walnut, the mail system for Cedar, provides facilities to send and retrieve mail and to display and classify stored (previously retrieved) messages. Walnut uses the Cypress database system [5] to maintain information about stored messages.

†83 The observant reader may have noticed that the date in the change log entry in the file I edited (Figure 28) is September 1, whereas the date in the Walnut control panel and on many of the messages is September 20. There are other such anomalies later in this paper, all attributable to the fact that the figures in this paper were not all produced on the same day.

found" or "Tennis Tickets Gone"). Messages regarding events I want to be sure not to forget, such as a talk or meeting, I can enter into my personal calendar/reminder system by simply clicking the Remind menu button on the corresponding message, as I have in Figure 30. The Reminder system obtains the time and date for the corresponding event from the message itself,†84 and when the corresponding time rolls around, a blinking icon is automatically displayed on my screen (see Figure 33). If I open this icon, the corresponding viewer will contain this message.†85

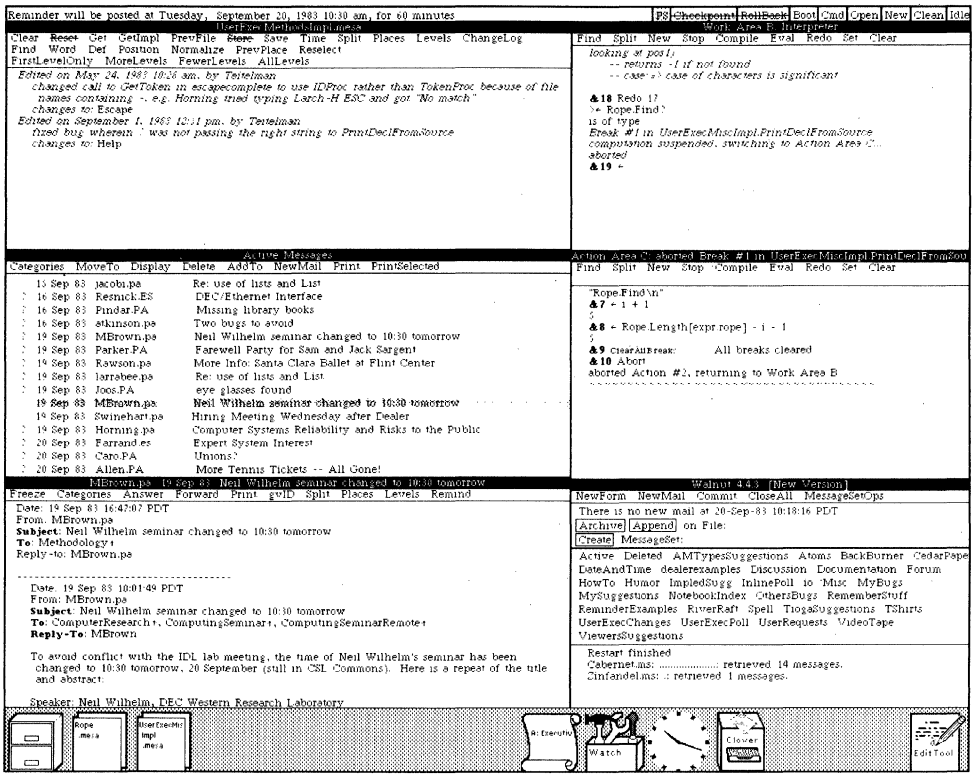


Figure 30

Entering a message into my reminder system

†84 You can see the feedback from the reminder system in Figure 29 in the message window at the top of the screen: "Reminder will be posted at Tuesday, September 20, 1983 10:30 am for 60 minutes." This time was computed from the string "10:30 tomorrow" in the subject field of the message, using the date field of the message to determine the reference point for "tomorrow," i.e., pretend today is 19 Sep 83 when figuring out what tomorrow is, even though I am actually reading the message on September 20 (the day after it was sent).

†85 Here is an excellent example of what we mean when we say Cedar is *integrated*: the various facilities can use each other in important ways since they all *coexist in the same address space.* (Here the reminder system uses both Walnut and Tioga.) Furthermore, there need not be any explicit context switch and corresponding loss of state when switching between tasks or programming tools, for example, in switching from debugging, to editing, to reading mail. Integration is one of the reasons why a large virtual address space (≥ 24 bits) was one of the Priority A items in our original Catalogue of Programming Environment Capabilities [8].

Messages that I want to save so that I can refer to them later I frequently sort into various categories called *message sets.*†86 I have about thirty of these categories and can add more whenever I need them. My current message sets are shown in the Walnut Control Panel (at the lower right in Figure 30): BackBurner, CedarPaper, Discussion, Documentation, etc. Notice in Figure 29 that my Active message set contains a number of messages about the use of lists. These are in response to a poll I sent to Cedar users about how they used lists (for material for this paper). In Figure 31, I have created a new message set called Lists, and am in the process of moving these messages into that message set so that I will have them all in one convenient place. I do this by pointing at the corresponding message in my Active message set viewer and then clicking the MoveTo menu button.

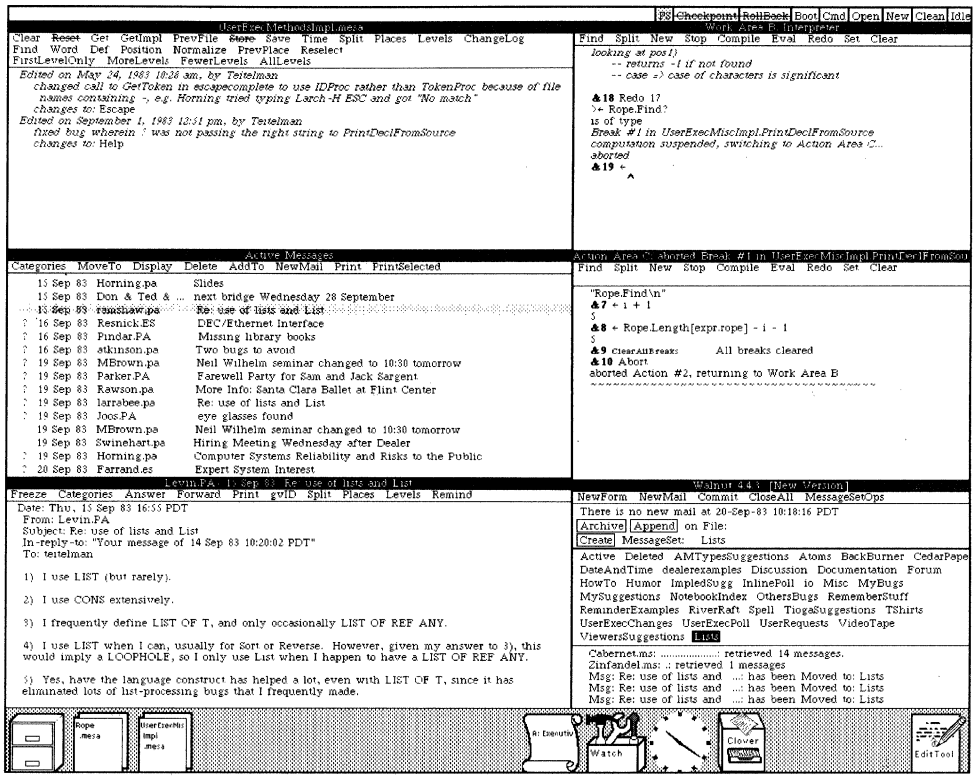


Figure 31

Sorting messages into message sets

If I point at one of the message set buttons in the Walnut control panel and click the mouse, Walnut creates a viewer on the corresponding message set. This viewer shows the date, sender, and subject of each message in the set. For example, I typically save messages about bugs in my software in the message set called MyBugs. The message regarding the bug I want to fix is in this message set, which I'll now open.

†86 Walnut's database contains two types of entities: messages and message sets. A message entity corresponds to a message retrieved from the mail distribution system. A message entity can be a member of one or more message sets. There are two distinguished message sets: Active and Deleted. A newly retrieved message automatically becomes a member of Active. A message that is removed from all other messages sets is added to Deleted. The user can create or destroy additional message sets as the demonstration illustrates.

The message that I am interested in is 11-Feb-83 Willie-Sue.pa bug??.†87 I'll click it, and Walnut will obtain the message contents from the data base and put it in a new viewer, as shown in Figure 32 (left column, top viewer).

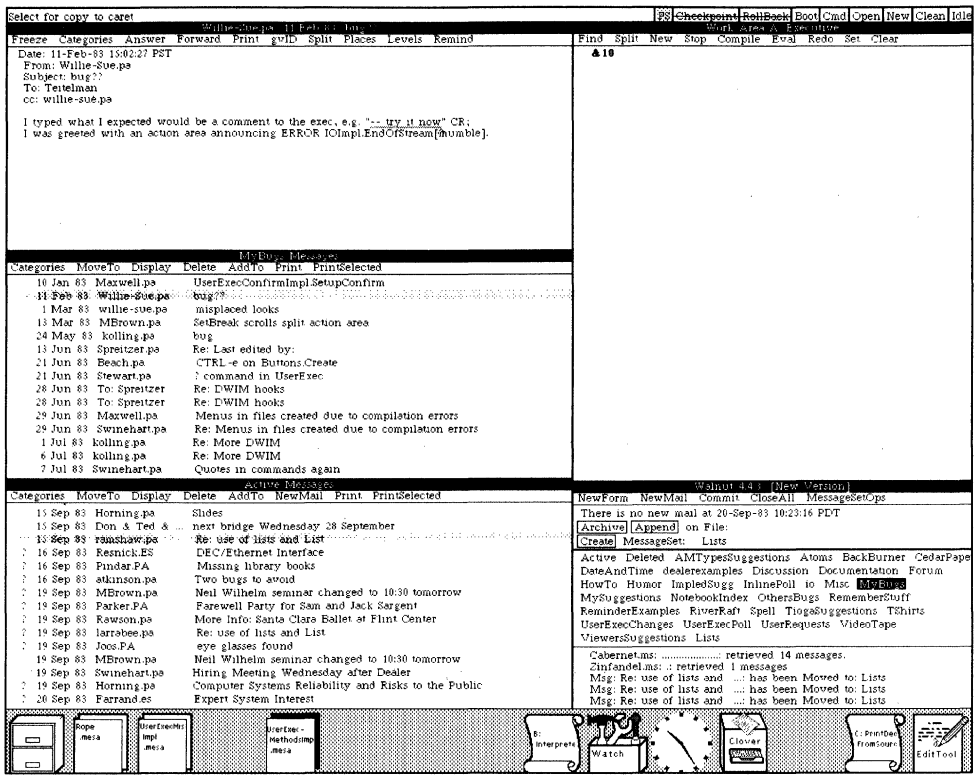


Figure 32

A user reports a bug via an electronic message

A Bug Report

The message states that when an event consisting of just a comment is typed to the executive, an error occurs. Let's try it and see. I'll return to Work Area A and then, instead of typing the comment, copy it from the message directly into the Work Area. In Figure 32, I have selected the corresponding characters in the message with the SHIFT key depressed. When I lift the SHIFT key, the characters will be copied into Work Area A.†88

†87 The video tape that this demonstration was taken from was originally produced in February 1983, whereas this paper was written in September 1983. Obviously this and other bugs that I will fix during the course of this demonstration were actually taken care of many months ago. However, for the purposes of this paper, I have restored Cedar to the state that it was in February, at least with respect to these changes, and am reenacting the scenario.

†88 Note that in this case I will be copying characters from a Walnut message viewer into an Executive Work Area, still using the same method as we used previously. Consistency of user interface!

&10 -- try it now
ERROR IOImpl.EndOfStream from InputImpl.GetCedarScannerToken1
computation suspended, switching to Action Area E...

(and down below a new Work Area pops up in which appears:)

Action # 1 (kind: signal, process: 204B) (from Work Area A)
ERROR IOImpl.EndOfStream[stream: {15501066B - Input From Rope Stream}] from InputImpl.GetCedarScannerToken1

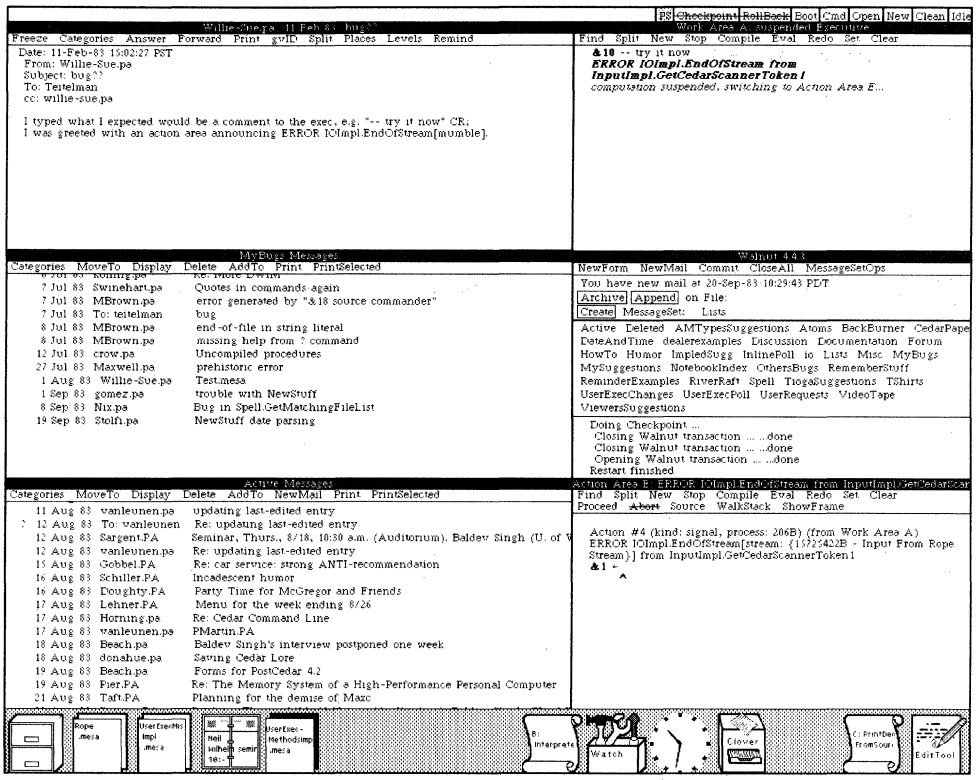


Figure 33

An error causes the creation of a new Action Area

Well, it's just like the message said. We got an EndOfStream error, and are now in a new Action Area.†89 †90 Let's walk the stack and see what's happening.

†89 Uncaught errors and signals are handled the same as breakpoints: they constitute actions and are given their own Action Area.

†90 Notice that, since it is now 10:30AM, the reminder concerning that talk I wanted to attend (entered in Figure 30) has popped up at the bottom of my display, fourth icon from the left. (Though the reader obviously can't see it in the figure, the icon is blinking to call itself to my attention.) If I were to open this icon, I would find the original message.

&1 WalkStack InputImpl.GetCedarScannerToken
&2 WalkStack InputImpl.GetCedarToken
&3 WalkStack InputImpl.fromTokenProc
&4 WalkStack InputImpl.GetCedarScannerToken
&5 WalkStack InputImpl.GetCedarToken
&6 WalkStack UserExecImpl.IsWellFormed

The first five levels of procedure nesting correspond to internal calls within the IO package. However, the procedure UserExecImpl.IsWellFormed looks more promising. Let's look at its source.

&7 Source userexecimpl.mesa 23932
IF Rope.IsEmpty[rope] OR NOT Rope.Equal["←", IO.GetCedarToken[stream]] THEN GOTO Yes; †91

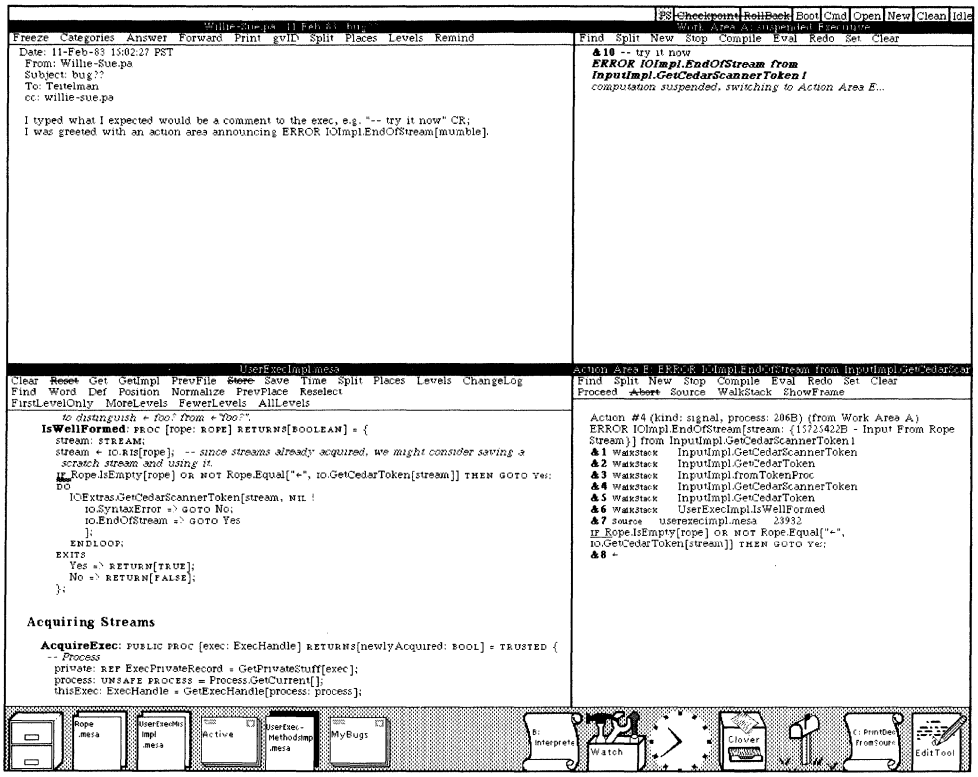


Figure 34

Tracking down a bug

The underlined location marks the place in the source that corresponds to where the computation is now. It looks like the program is using the procedure IO.GetCedarToken to read a token from stream. Let's examine the variable stream using the interpreter.

†91 The Cedar Language provides for an extremely restricted form of GOTO statements, namely to a series of labeled statements called an ExitsClause that appear at the end of a block. Think of GOTO as the Cedar way of spelling EXIT.

```
&8 ← stream
{15725422B - Input From Rope Stream}
```

Note that streams have PrintProcs which print out the kind of stream, suppressing the stream's actual representation.†92 In this case, we do want to look inside of the stream at its data, which we can do using the interpreter. First, I'll find out the stream's type using the interpreter.

```
&9 ← &?
is of type STREAM: TYPE = REF IO.STREAMRecord;
IO.STREAMRecord: TYPE = RECORD[streamProcs: REF IO.StreamProcs, streamData: REF
ANY, propList: Atom.PropList ← NIL, backingStream: STREAM ← NIL]
```

This says that a stream is a REF to a record consisting of four fields: streamProcs, streamData, propList, and backingStream, each of which have the indicated type. Let's look at the streamData field, which contains the data for this particular stream.

```
&10 ← &.streamData
↑[rope: "-- try it now", pos: 13]
```

Even though the type of this field is REF ANY (so that different kinds of streams can store different types of data in the same field), the interpreter is able to figure out the type of the referent using the run-time type system. It tells me that the data for this stream is a REF to a record consisting of two fields named rope and pos, whose values are "-- try it now" (notice that this ROPE has 13 characters), and 13. In other words, the current position, pos, does indeed correspond to the end of the stream. What happened to the previous 13 characters?

In puzzlement, I decide to look at the definition for IO.GetCedarToken. I select the characters IO.GetCedarToken in the source viewer, and then click the Open menu button to create a new viewer on the IO interface positioned at the definition of GetCedarToken, as shown in Figure 35.

†92 A stream in Cedar is simply a producer and/or consumer of byte sequences. The stream abstraction can be implemented in a variety of ways. For instance, the producer behind an input stream might be a file or a user typing at a keyboard. We call each stream implementation (file, keyboard, and so on) a *stream class*. One of the most important aspects of streams are that a client program can manipulate a stream without regard to the class that implements it. Thus, varying stream implementations can be substituted without effect on the program. Furthermore, new implementations of streams can be supplied by the user at runtime. Examples of such user defined streams are: decrypted input and encrypted output streams layered on top of other streams, an output stream that automatically indents to indicate structure, a stream which reads Intel format absolute binary object files, and a stream that emulates Unix pipes.

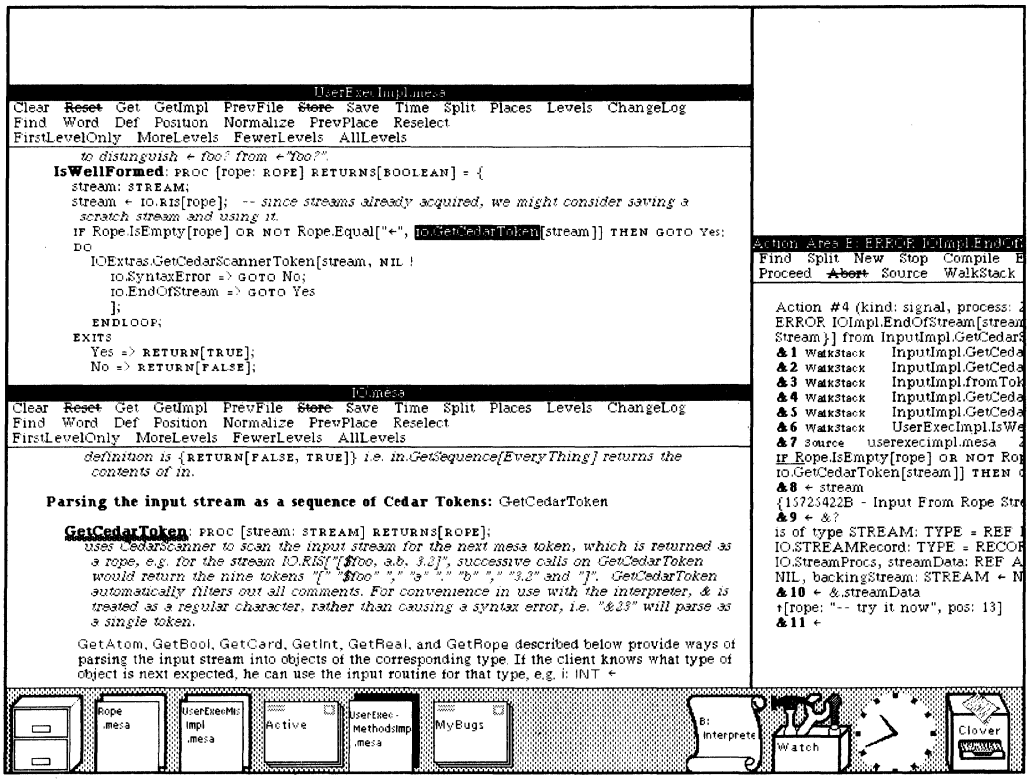


Figure 35

The IO interface serves as online documentation

Aha! The comment in the IO interface says: *"GetCedarToken automatically filters out all comments."* The problem is that when my program asks for the next token from the stream, there isn't one, because comments are filtered out when reading tokens. So the error EndOfStream is raised.†93 What I should be doing in this program is catching the signal EndOfStream in the call to IO.GetCedarToken, and simply returning TRUE. Let's make that change.

Now let's return to our Action Area on the right, and since we are finished with this problem, we can abort the action, and return to the Work Area above.

†93 The Cedar language uses signals and errors as a mechanism for handling exceptional conditions. (The only difference between a signal and an error is that the program that catches the signal can resume the program that raised the signal, whereas errors cannot be resumed.) Think of a signal/error as a procedure call where the body of the procedure is determined dynamically using the call stack. In this way, an implementor can allow a client to specify what to do for various exceptional cases, without requiring that the client specify a plethora of extra arguments to cover all such cases, or returning various invalid values for which the caller must check. This way of handling exceptions has two important aspects, one for the human reader of the program, and one for its execution efficiency. First, anyone reading the program can see immediately that an exceptional condition can arise by the catch phrase, knows that this is an unusual event, and can read on with the normal program flow. Second, when the program is executing, the code to handle the exceptional cases is not executed on every call, but only when the actual signal is generated, i.e., when the exceptional condition occurs [20]. It is worth pointing out that a facility for raising and catching signals very similar to Cedar's has recently been implemented for Interlisp.

Compiling, Support for Concurrent Operations

Now I want to compile the files that I have edited. The system keeps a list of those files that need to be compiled, i.e., those that were edited but not yet successfully compiled. It also provides visual reminders in the form of a black border around the corresponding icons, as shown in Figure 36. I can instruct the system to compile all of the files that need compilation via the command **compileall.**[†94]

```
&10 compileall
>Compile UserExecMethodsImpl.mesa UserExecImpl.mesa
```

While that is going on, I'll answer Willie-Sue's message. I click the **Answer** menu button in the viewer containing her message, and Walnut creates a reply form containing the appropriate Subject, To, and cc fields. In Figure 36, I am in the process of composing my answer in the viewer on the lower left,

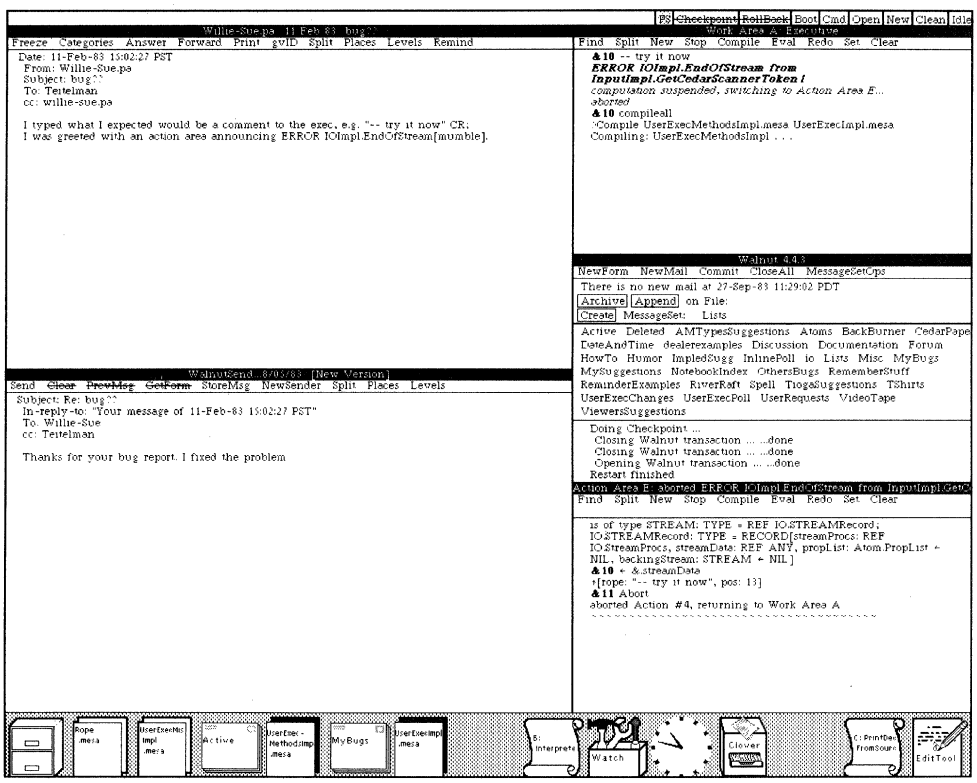


Figure 36

Concurrency: answering mail while compiling

†94 CompileAll simply keeps track of those files that have been edited. It does not deduce that because Interface A has been recompiled, Modules B, C, and D also need to be recompiled. This latter behavior is much more ambitious and falls under the category of what we call System Modeling: "The user describes his software in a system model that lists the versions of files used, the information needed to compile the system, and the interconnections between the various modules. The modeler is connected to the editor and is notified when files are edited and new versions are created" [26]. A preliminary version of a system modeler has been built and tested, and a more comprehensive version has been partially implemented.

while the compiler continues to run in the Work Area at the upper right.†95

I finish composing the message, and click Send, and the message is sent on its way. In Figure 37, the Walnut Control Panel tells me that the message has been delivered. The next time that Willie-Sue clicks her NewMail menu button, she will see the message.

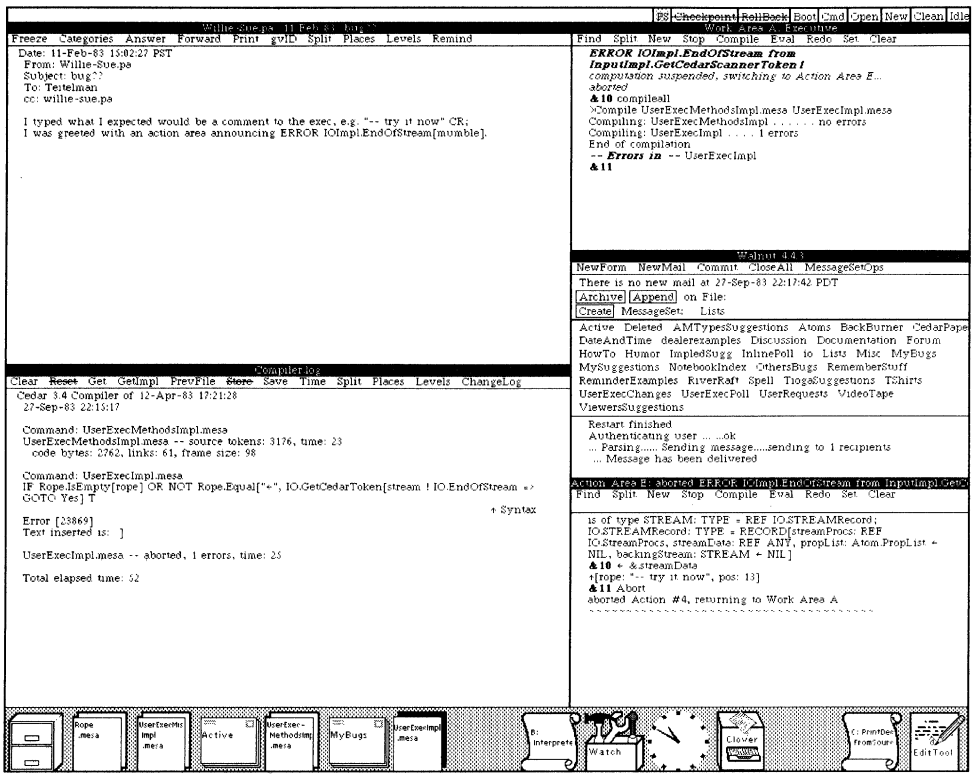


Figure 37

Compiler error log

Meanwhile, the compiler has successfully compiled the first file (notice in Figure 37 that the black border around the fifth icon from the left is now gone), but ran into a problem in compiling the second file. The UserExec has created a viewer on the left which displays the compiler log containing the error message.

†95 As mentioned earlier, Cedar supports and encourages concurrent operations, and users make heavy use of this parallelism. Here I am sending a message while compiling a file. In this particular case, only one task requires my attention: the other is running in background (my background, not the computer's). However, it is not uncommon for users to be performing several foreground tasks simultaneously, such as editing several source files at the same time, or debugging a program by stepping it from breakpoint to breakpoint, while simultaneously reading mail, etc. The important point is that the user's interactions with the system can match the style with which he is most comfortable.

The error is a simple syntactic error, a missing '].†96 I'll make this fix and recompile. In the meantime, this reminds me that a user had sent me a message about a request concerning the UserExec's behavior with regard to the compiler log. I keep such messages in my UserRequests message set. I click the UserRequests button in my Walnut Control Panel to create a viewer for this message set, and then click the corresponding message in the message set viewer (see Figure 38).

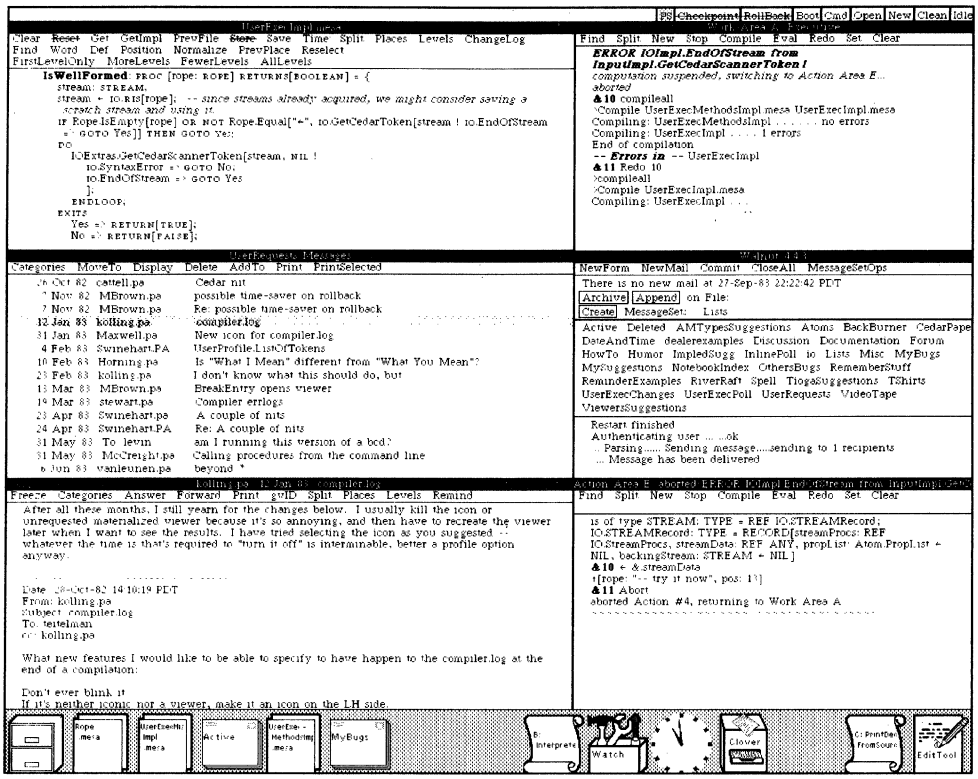


Figure 38

A user request

The User Profile

The message (bottom viewer, left column in Figure 38) states that the user wants to be able to specify that compiler logs are always created iconic instead of being open, as in Figure 37. Since some users like the way compiler logs currently work, to satisfy this user's request, I am going to define a new user profile option so that each user can specify how they want the compiler log handled. In the area of user interface, rather than enforcing a consensus upon everyone, we allow individuals to tailor the system to suit themselves, enabling facilities that they like and disabling those that they don't. For example, I'll open my profile.

†96 The compiler log includes for each error a position (character count) in the source file, e.g., in Figure 37 the Syntax Error occurred at position 23868. The user selects this position, and then clicks the Position menu button in the corresponding source file and the source file is automatically positioned at the indicated location (as I have done in Figure 38). The user can thus quickly step through the source file from error to error and make the necessary edits. Even so, the process of getting a file to compile successfully is still very tedious. More tools are required. For example, many compiler errors turn out to be of the nature that their correction could be automated. One could imagine an extension of DWIM that would handle this task.

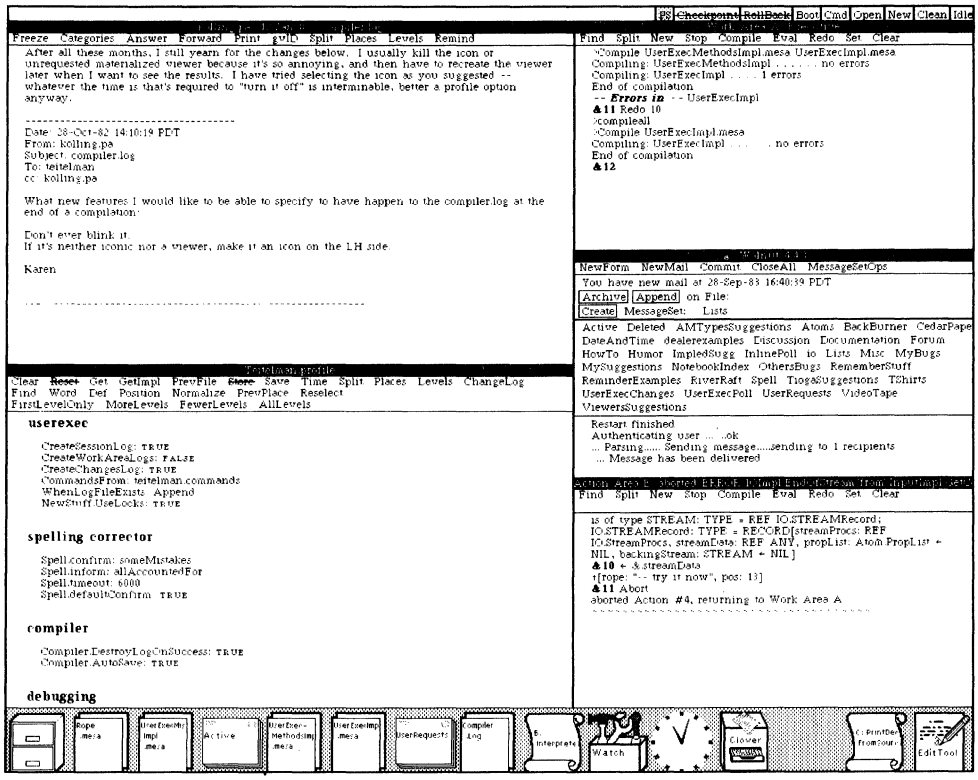


Figure 39

User profiles allow tailoring of the system to suit individual tastes

As you can see (lower left in Figure 39), my profile specifies a variety of options and defaults.

Let's implement the new profile option. I'll create a new viewer, load it with the appropriate source file, and then scroll to the procedure ShowLog, where I want to make the change. What I want to do is to insert a conditional statement that will check the user's profile to determine whether or not to create the viewer for the compiler log iconic.

Abbreviation Expansion and Templates as an Aid for Editing Programs

To accomplish this, I will use Tioga's abbreviation expansion facility to cause a template for an IF-THEN statement to be inserted. To do this, I type IF followed by CTRL-E (E with the CTRL key depressed). This causes Tioga to expand the abbreviation for IF into the template you see in Figure 40.†97 This template contains two fields, TEST and TRUEPART, each delimited by special brackets called *placeholders*, which are displayed as ▶◀. Tioga allows me to move to the next/previous field delimited by placeholders with a single keystroke. If I am positioned at one of these fields, anything I type automatically replaces the field. In Figure 40, I am ready to specify the predicate for my IF-THEN statement.

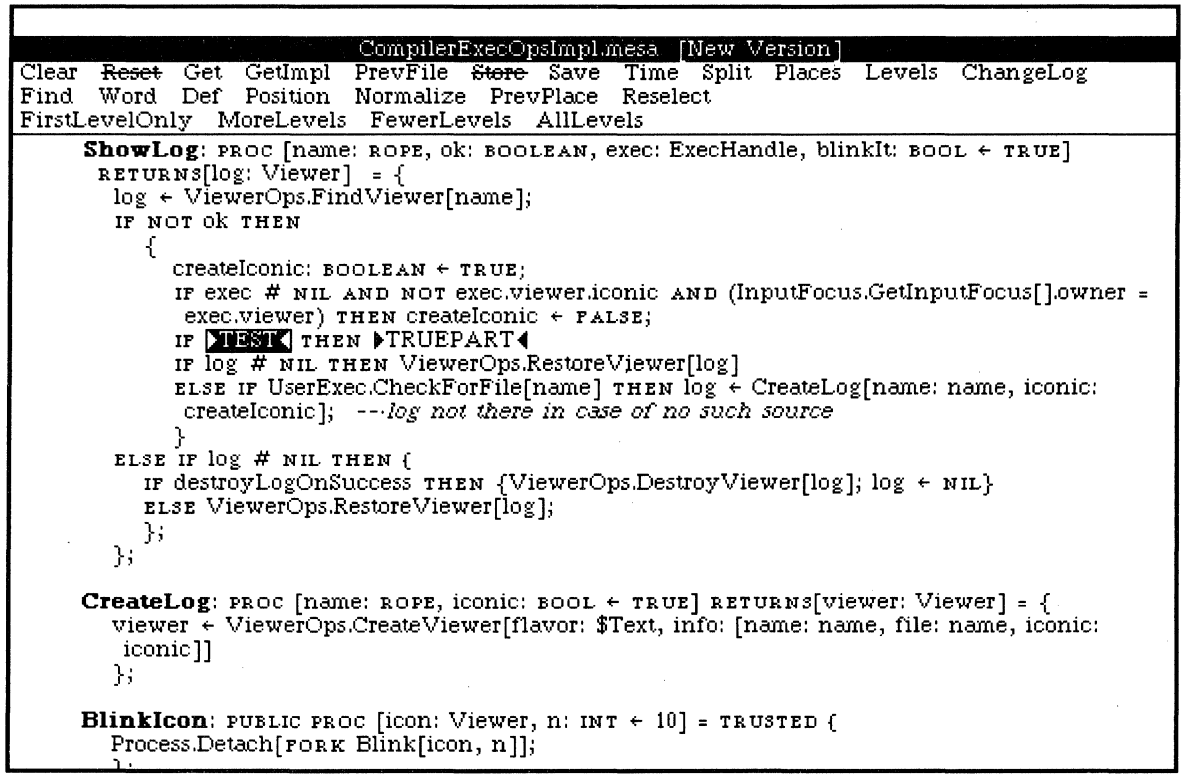


Figure 40

Tioga abbreviation expansion facility

†97 The Tioga abbreviation expansion facility helps the user in dealing with the Cedar syntax, avoiding errors, and formatting programs consistently. There are similar abbreviations for many of the language constructs in Cedar, e.g., FOR expands to FOR ▶ControlVariable◀ ← ▶InitialExpr◀, ▶NextExpr◀ DO ▶BODY◀ ENDLOOP. In addition, the user can add to or change the set of predefined abbreviations.

The predicate I want to use is the procedure UserProfile.Boolean. I type the name of the procedure, UserProfile.Boolean, and then I type CTRL-E again, this time to request a template for its arguments. Note that UserProfile.Boolean is not defined as an abbreviation; Tioga *computes* a template consisting of the names, types, and default values for this procedure using the run-time type system, and inserts it in the document as shown in Figure 41.†98

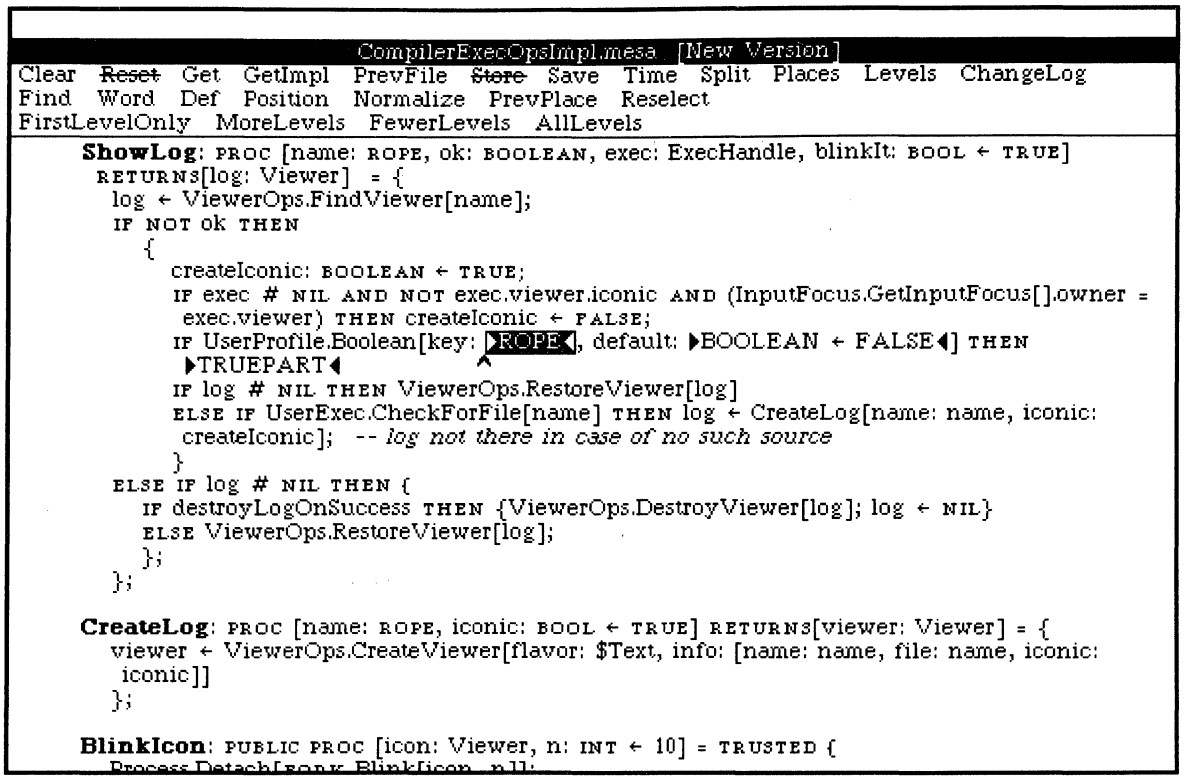


Figure 41

Computing a template for a procedure call

As the template indicates, UserProfile.Boolean takes two arguments; the first is named key, and is of type ROPE, the second is named default, and is of type BOOLEAN. I'll call the key for the new user profile option that I am going to define Compiler.IconicLogs. If the value of this key is TRUE, i.e., if the user's profile contains an entry of the form Compiler.IconicLogs: TRUE, then we make the compiler log viewer iconic (by setting the variable createIconic to TRUE). The entire statement that I inserted is:

IF UserProfile.Boolean[key: "Compiler.IconicLogs", default: FALSE] THEN createIconic ← TRUE;

but I only had to type the underlined characters plus two CTRL-E's.

†98 Runtime availability of all source program information was one of the priority B items on our original catalogue of programming environment capabilities. Underlying this was our desire to make it easy to extend the set of tools for assisting the programmer. The computed template facility shown here is a good example of the kind of thing we had in mind.

Using the Interpreter for Experimentation

There was another request in my UserRequests message set concerning compiler logs, namely that the compiler log use the typescript icon rather than the document icon, to make it easier to distinguish the compiler log from other iconic Tioga documents. If we look at all of the icons at the bottom of Figure 39, we can indeed see that it is hard to find the compiler log among all these other documents with the same icon (it's in the center).

Before we make this edit, let's try changing the icon for this viewer by hand, i.e., by using the interpreter.†99 So I'll reopen my interpreter Work Area, select the compiler log, and use the Eval menu button to evaluate the current selection (see Figure 42).

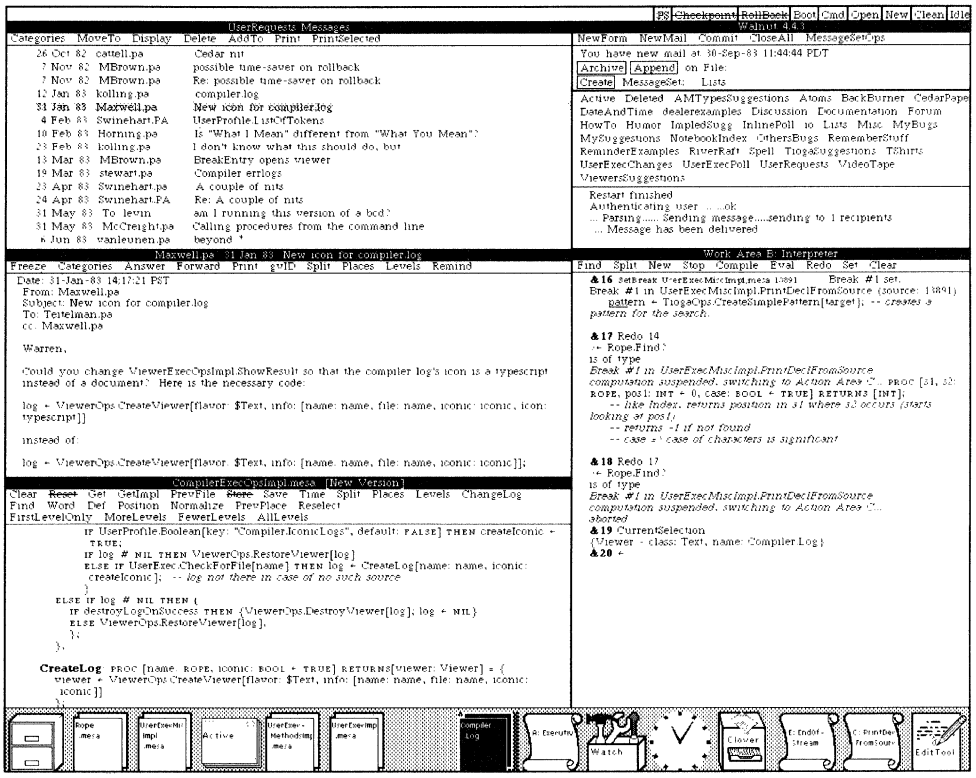


Figure 42

Using the interpreter to experiment

†99 Using the interpreter to try things out before going to the trouble of making changes to a program is a technique that is relatively new to the Mesa community (although it has been commonplace in Interlisp for many years). Part of the reason for this is historical. All levels of the Mesa system are written in Mesa itself, even the lowest level of run-time support, i.e., there is no assembly code or other language. Since using source-level debugging was desirable for the entire system, even those primitives that the debugger itself would need to operate, the solution adopted was to implement a non-resident or *world-swap* debugger, one in which the debugger operated at arm's length from the debuggee in an entirely separate address space. Interpreting expressions in this remote world was slow and cumbersome. Furthermore, the interpreter only handled a limited subset of the language.

We had higher aspirations for Cedar: to provide a resident debugger, one that shared the same address space as the programs being debugged, as well as a complete expression interpreter. (A world-swap debugger is included in Cedar, however, for debugging those levels of the system that are more primitive than the debugger itself.) Thus, the Cedar environment represents the first opportunity that Mesa programmers have had for using an interpreter to carry out experiments. Consequently, this style has not yet caught on.

```
&19 CurrentSelection
{Viewer - class: Text, name: Compiler.Log}
```

The value of this event is the viewer for the Compiler Log. I can manipulate this value. For example, let's look at its icon field.

```
&20 ← &19.icon
document
```

As expected. Now let's change this field to be typescript. The simplest way to do this is to repeat the previous line up to but not including the CR. I can do this with a single keystroke (by typing ESC). Now I'll complete the line by typing an assignment that will assign a new value to the icon.

```
&21 ← &19.icon ← typescriptt
typescriptt -> typescript
typescript
```

Obviously, I am making some of these typing mistakes just to demonstrate the pervasiveness of the error-correction facilities. However, this correction is especially interesting because, in this case, DWIM uses as candidates for the correction only the set of values that an object of type icon can assume. In order to find what these are, DWIM uses the run-ime type system to *compute* this information when the error occurs. In this way, DWIM can work on user defined types as well as those that are defined in the basic system.

Now let's repaint the icon and see how it looks. I can do this by using the procedure PaintViewer, which is in the interface ViewerOps.

```
&22 ← ViewerOps.PaintViewer[&19]
***Missing Arguments: hint: ViewerOps.PaintHint
```

What's a paint hint? I'll evaluate it and find out.†100

```
&23 ← ViewerOps.PaintHint
ViewerOps.PaintHint: TYPE = {all, client, menu, caption}
```

This says that a PaintHint is an enumerated type consisting of the four values all, client, menu, and caption. I'll bet I can just pass in all for the hint argument. However, there may be other arguments to PaintViewer that also have to be specified, so I'll use the template feature again to construct a template for PaintViewer, as shown in FIgure 43.

---

†100 Note that in this example, we are evaluating an expression whose value is a *type*. One of the goals of Cedar was to make types into first-class citizens. We have not succeeded in making much progress on this so far with respect to the Cedar language; it is not possible to pass types around as values, and there is no polymorphism in the language. However, with regard to the Cedar run-time system, it is possible to perform a wide variety of operations on types, e.g., given a record type, compute the names and types for each field, or, given a REF type, compute the type of the referent. Here is an example of an application: a user tried to use the interpreter to call a procedure one of whose arguments was defined to be of type LIST OF ROPE. For this argument, he typed LIST["abc", "def"], a list of ropes, but got an error message complaining that the value supplied was of type LIST OF REF ANY. This is a current shortfall in the interpreter: whenever it sees LIST, it constructs a LIST OF REF ANY regardless of the target type. However, it was fairly straightforward to extend DWIM to perform the necessary coercion after the fact. This involved writing a procedure which took a type, TYP, and a value, VAL, and: (1) determined whether TYP was of the form LIST OF REF T for some type T, (2) if so, determined the type of the elements of the list, i.e., REF T, (3) checked that VAL was of type LIST OF REF ANY, (4) if so, iterated down VAL and verified that each element was in fact of type REF T, and finally (5) constructed a new value of type LIST OF REF T whose elements were the corresponding elements of VAL.

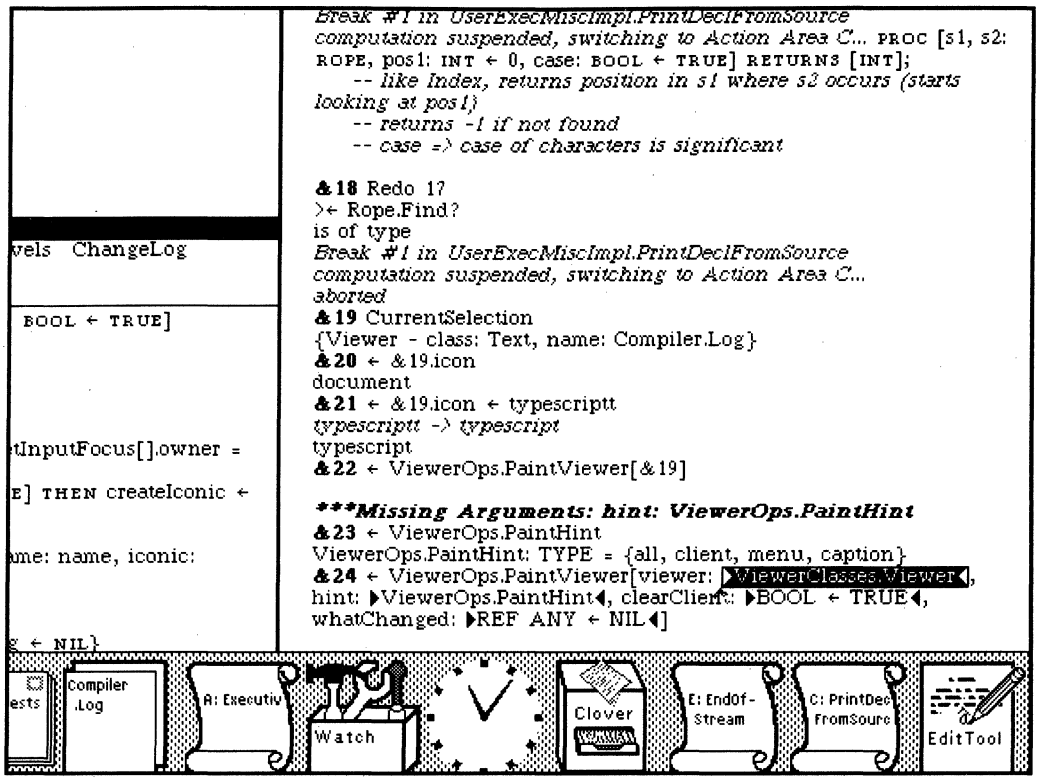


Figure 43

Computing a template for a procedure call to be executed

There are two additional arguments, clearClient and whatChanged, but both have default values so I can omit them. I fill in the viewer field with the viewer I want to repaint, which is the value of event 19, fill in the hint field with all, and I'm done.†101

```
&24 ← ViewerOps.PaintViewer[viewer: &19, hint: all]
{does not return a value}
```

†101 It is interesting to consider how few keystrokes and mouse actions were actually necessary to construct the expression: ViewerOps.PaintViewer[viewer: &19, hint: all]. First, I selected the word "ViewerOps" in event 22 by clicking in that word using the middle button on my mouse; then I extended the selection to include "PaintViewer" by clicking in the latter word using the right button. Both of these selections were performed while holding the SHIFT Key down. I then lifted the SHIFT key causing the entire selection, i.e., the characters "ViewerOps.PaintViewer," to be inserted into the Work Area. Next I typed CTRL-E to construct the template shown in Figure 43, SHIFT-selected the word "&19" from event 22, hit the NEXT key on my keyboard to move the caret to the hint: field; SHIFT-selected "all" from the value of event 23; and then finally hit CTRL-NEXT to delete the remaining argument fields, all of which have default values, and to advance the caret to just past the "]." I completed the event by typing CR. Total: 4 mouse clicks, 4 keystrokes.

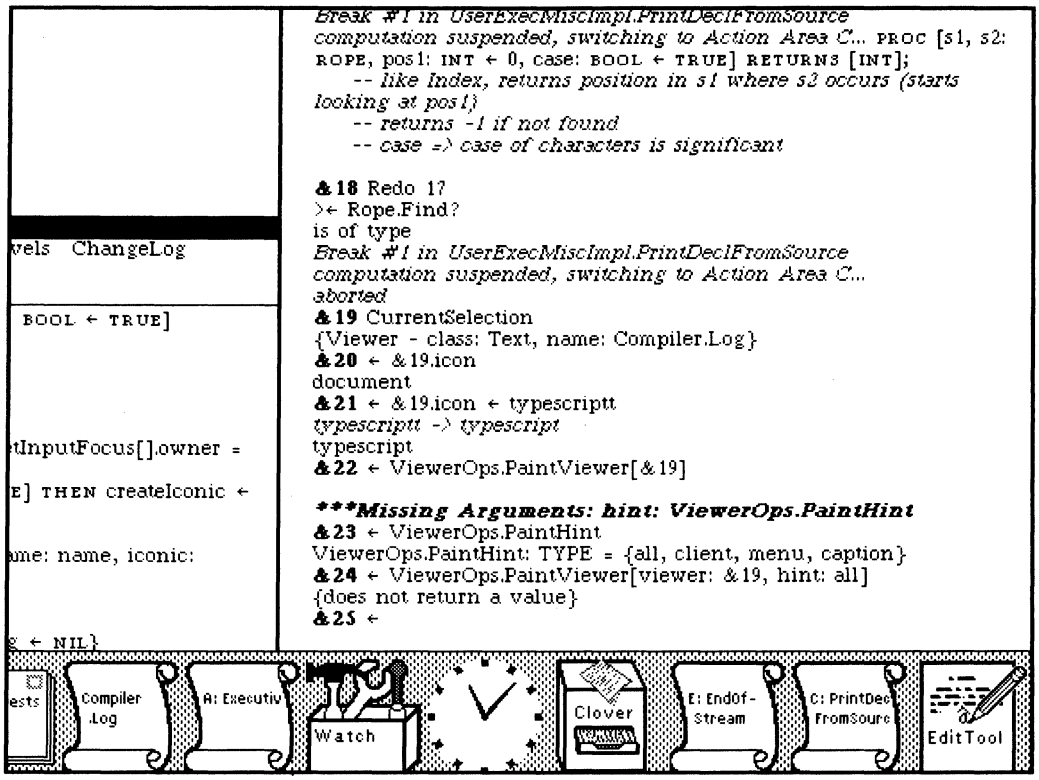


Figure 44

The icon for the Compiler Log has been changed to a typescript

Sure enough, the icon for the compiler log is now a typescript. I can now implement this feature by editing the source, which is still in the viewer immediately to the left.

However, now I find that while I have made it easy to distinguish the compiler log from the other documents, there are so many typescript icons that it is hard to find the compiler log among all of them (Figure 45).

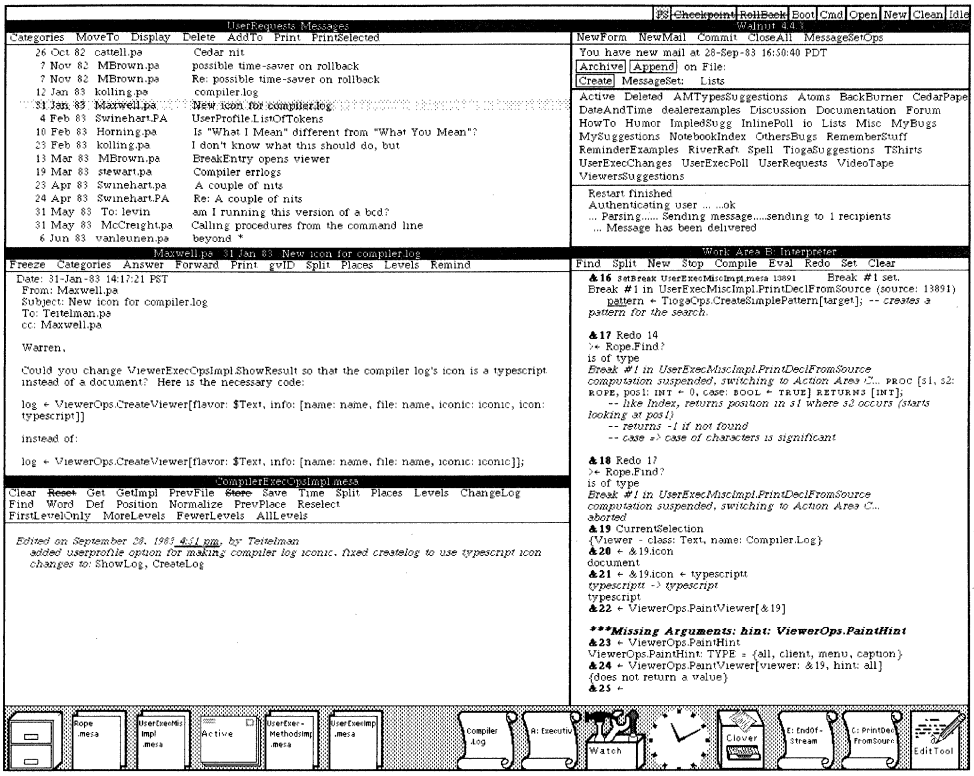


Figure 45

Now there are too many typescript icons

Designing a New Icon

One way of solving the problem of too many typescript icons is to use a different icon for some of the executives. I'll use a graphics tool called the Icon Editor to design a new icon for Action Areas executives (the two executives on the right in Figure 45).

```
&25 run iconeditor
Loaded and started: IconEditor.bcd
```

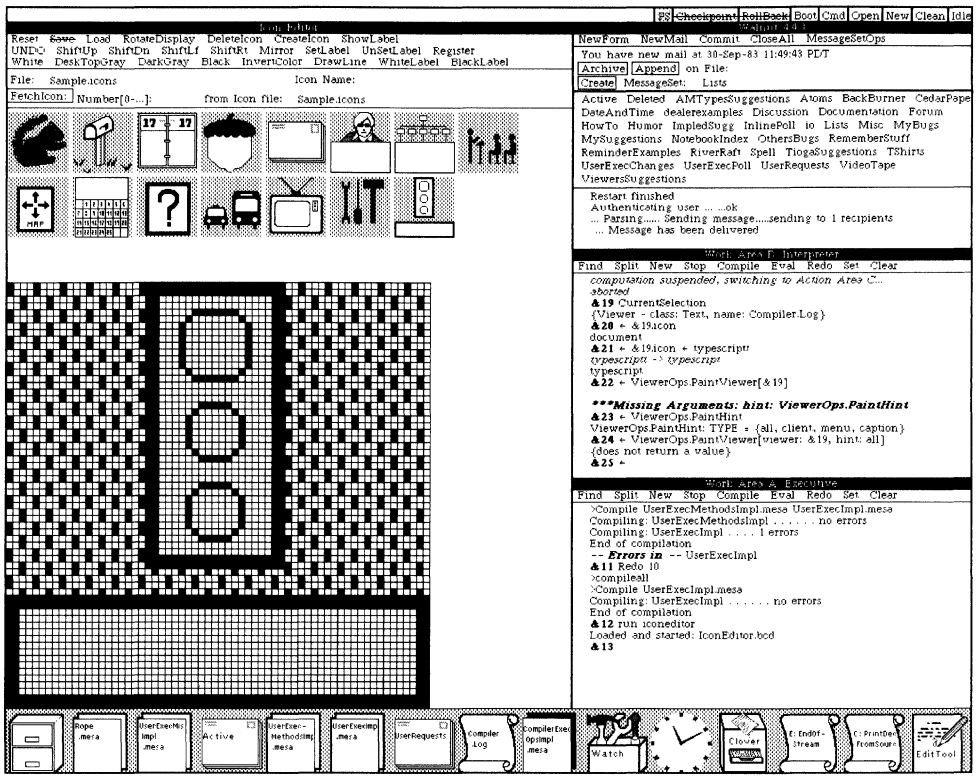


Figure 46

The Icon Editor

In Figure 46, the Icon Editor contains some of the icons that other people have designed for various applications. The Squirrel icon is for our data base facility which is named Squirrel. Next to it is the Walnut mail reader icon you have already seen. Also included in the two rows of icons are an icon for a calendar, a bus schedule, a TV listing, an organization chart, etc. The last icon in the second row is the trafficLight icon I am working on (for executives stopped because of a signal).

The 64 x 64 array of squares that occupies the lower two-thirds of the Icon Editor's viewer represents the individual pixels in the icon currently being edited. I can change individual pixels from black to white or vice versa by clicking with the mouse in the corresponding square. I can also draw lines, change rectangular areas to different textures (stiple patterns), shift rectangular areas up, down, left or right. As I make changes in this array, the smaller version of the icon is updated so that I can see how the icon is going to look, actual size. You can see this as I make a few finishing touches to my icon–darkening the red light and adding rays of light coming from it.

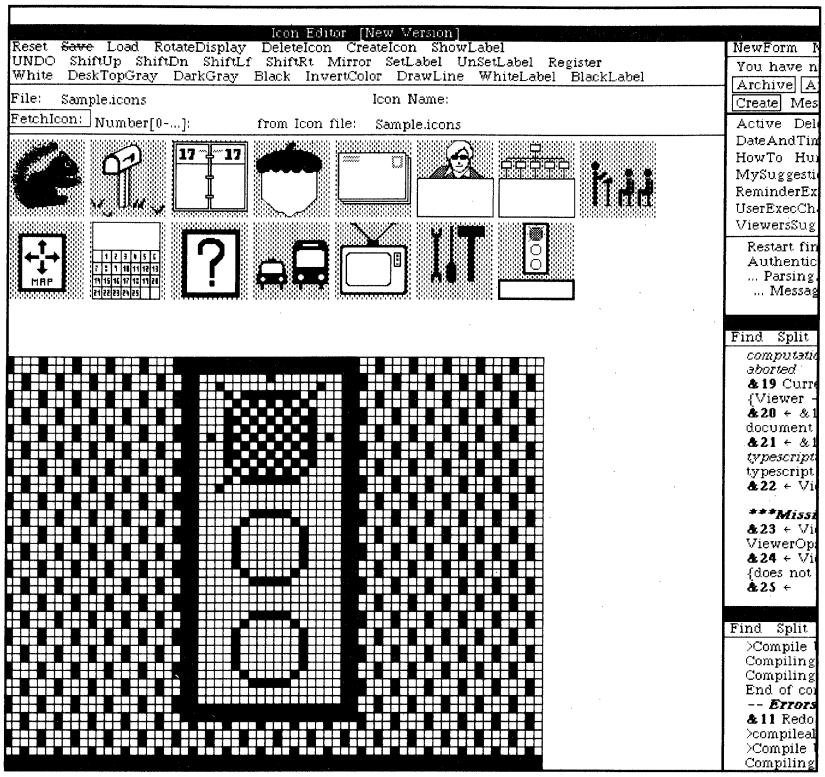


Figure 47

Designing a new icon

I'm happy with the icon now, so I'll save it on a file. I'll also associate the name "trafficLight" with this icon by using the Register menu button IconEditor viewer. This will allow me to refer to the icon by name without having to remember where it is stored.

Now let's use the interpreter to change the icon of one of my executives to be the trafficLight and see how it looks. First, we obtain an exec handle using the same method we did earlier, namely selecting the viewer and evaluating the current selection.

```
&26 ← CurrentSelection
↑[viewer: {Viewer - class: Typescript, name: Action Area E: aborted ERROR
IOImpl.EndOfStream from InputImpl.GetCedarScannerToken1}, privateStuff: 7246044B↑]
```

This value is the handle for Work Area E. Now let's set its icon to be a trafficLight.

```
&27 ← &26.icon ← IconRegistry.GetIcon["trafficLight"]
selection failed on icon
```

The viewer is one of the fields of the exec handle, and icon is one of the fields of the viewer. I am one level of indirection off: I should have said "&26.viewer.icon." Since what I did type is correct in every other respect, I can fix this error by simply replaying the line by typing ESC, pointing at the "." in "&26.icon,", and typing "viewer," as I am in the process of doing in Figure 48.†102

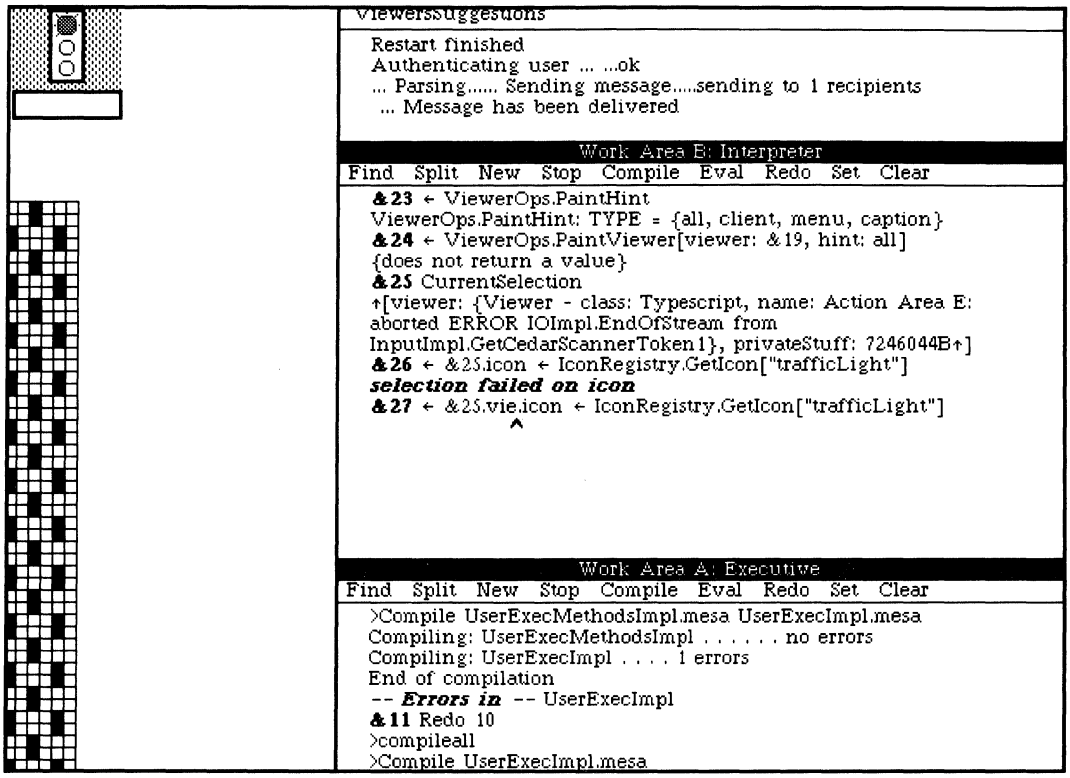


Figure 48

Editing events as they are being composed

&28 ← &26.viewer.icon ← IconRegistry.GetIcon["trafficLight"]
27B? †103

†102 In previous examples, the caret was always at the end of the typescript, so that our editing consisted of simply appending characters. This example illustrates that we really can edit, in the full generality of the term, events that are being entered for execution.

†103 The type of an icon is somewhat esoteric: it is a MACHINE DEPENDENT enumerated type. The predefined icons such as document, typescript, tool, etc., have names for the corresponding values, and hence print nicely. However, user-defined icons print in the strange fashion you see here. Ignore it.

Now let's repaint the icon for Action Area E and see how it looks. We already have an expression in event 24 that is pretty close to what we want, namely ViewerOps.PaintViewer[viewer: &19, hint: all]. We can use the use command to specify reexecution of this event with a different value for the viewer argument.†104

```
&29 use "&25.viewer" for &19
>← ViewerOps.PaintViewer[viewer: &25.viewer, hint: all]
{does not return a value}
```

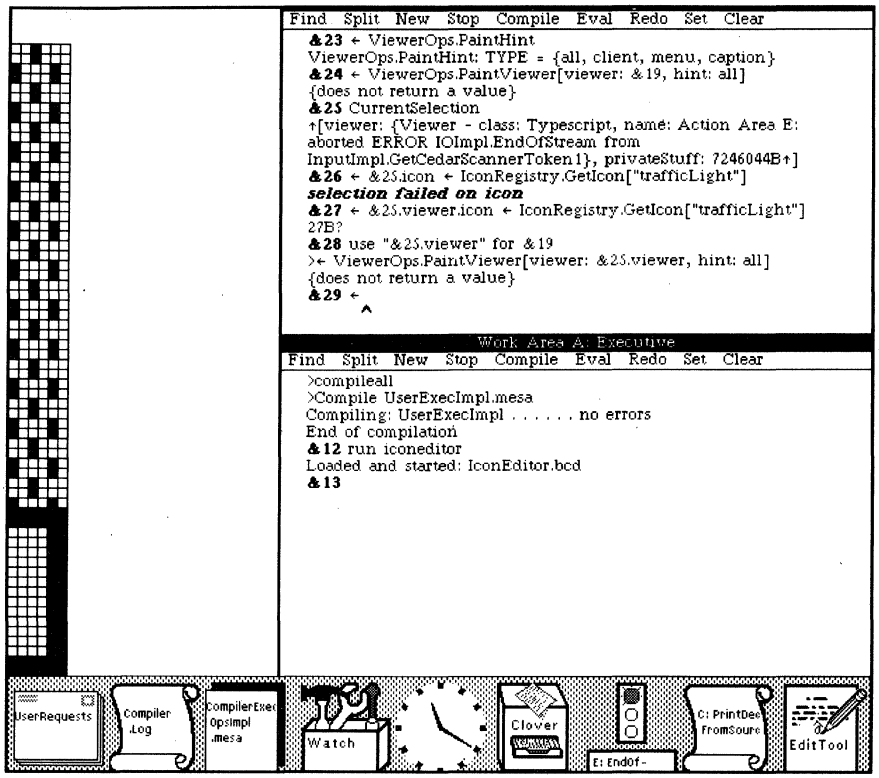


Figure 49

The Action Area icon has been changed to a traffic light

And there's our traffic light.

Let's go ahead and make the edit that will cause the system to use the trafficLight icon for Action Areas. We simply create a viewer on the file ActionAreasImpl, and insert at the appropriate place in the procedure NewAction the statement:
exec.viewer.icon ← IconRegistry.GetIcon["trafficLight"];
(The characters to the right of the ← can simply be copied from event 27.)

†104 We could also have replayed event 19 (either by SHIFT-selecting it or via the REDO command), and then made the desired change by editing. In this particular example, this would have actually resulted in fewer keystrokes than the use command. However, for a fast typist, moving one hand to the mouse and then positioning the mouse appropriately might actually take more time than typing the above. In any case, I wanted to demonstrate this feature of the history facility.

Wrapping it up

Now let's compile the files we have changed.

```
&13 compileall
>Compile ActionAreasImpl.mesa CompilerExecOpsImpl.mesa
Compiling: CompilerExecOpsImpl . . . . . no errors
Compiling: ActionAreasImpl . . . 1 errors
End of compilation
-- Errors in -- ActionAreasImpl
```

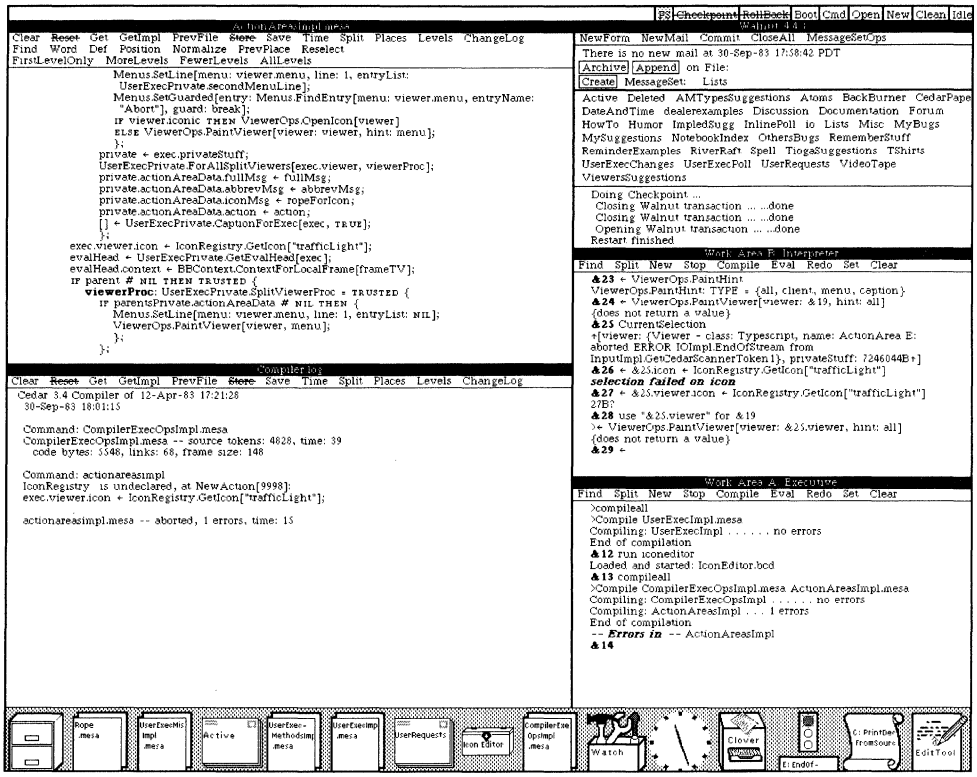


Figure 50

More simple compiler errors

There was an error in the compilation of the second file. It is displayed in the log on the left in Figure 50: "IconRegistry is undeclared." This error is due to the fact that I forgot to add IconRegistry to my Directory and Imports list and it reveals one area of weakness in Cedar common to many strongly typed languages: the tedium of getting a file to successfully compile. The majority of compiler errors turn out to be of the nature that their correction could be automated (such as the case with the missing '] earlier). An extension to DWIM to handle such errors automatically would be of great utility.

Let's fix this error and recompile.

```
&14 redo 13
>compileall
>Compile ActionAreasImpl.mesa
Compiling: ActionAreasImpl . . . no errors
End of compilation
```

The compilation has finished successfully. Now let's bind the program.

```
&15 bind userexecutive
Loading Binder.bcd...
Binding: userexecutive . . . . . no errors
End of binding
```

Unfortunately, since the changes we have made were to the UserExec, a component of the system that is already running, in order to test the changes we have to boot (reload) the system; we can't simply replace the UserExec that is running with the new one. Reloading of booting takes about two minutes. We hope to implement a facility for replacing an individual module in a running system. This should greatly improve the turnaround time on making and testing changes.†105

```
&16 boot
```

Testing Our Changes

(Later...) We have just finished booting the system. Let's test our changes. The system maintains a log of all of the files that were changed in our previous session. Let's open Work Area A and then open this log.

```
&2 open changes.log
Created Viewer: Changes.Log
```

The changes log tells us (see Figure 51) that the first problem we fixed was the one regarding not getting comments from files when I typed "?". Let's create an interpreter Work Area and try it out.

```
&1 ← Rope.Find?
is of type PROC [s1, s2: ROPE, pos1: INT ← 0, case: BOOL ← TRUE] RETURNS [INT];
   -- like Index, returns position in s1 where s2 occurs (starts looking at pos1)
   -- returns -1 if not found
   -- case = > case of characters is significant
```

†105 One of the top priorities in our original catalogue of programming environment capabilities was fast turnaround for minor program changes (< 5 sec). "Our concern with fast turnaround comes from the observation that programming should be *think bound*, not *compute bound*. There are several 'knees' (points of substantial non-linearity) in one's perception of response delays. One such knee is in the vicinity of 3 to 5 seconds. We believe that it is essential to reduce the system time for minor program changes to below this point" [8]. While the changes that we made in this tour were not minor, sad to say that even had they been minor, we would still have been forced to reboot (or at least return to an earlier checkpoint), in order to test them out. Attacking this shortcoming is now one of our highest priority items.

The UserExec printed the comments correctly (see Figure 51). The next change was to fix the problem wherein an event consisting of just a comment caused an EndOfStream error. Let's try this in Work Area A.

```
&3 -- try it now
```

That worked: it didn't raise an EndOfStream signal like it used to. The next change was to the procedure CreateLog to cause it to use the typescript icon for compiler logs. Let's try this out.

```
&2 ← CompilerExecOpsImpl.CreateLog["compiler.log"]
{Viewer - class: Text, name: compiler.log}
```

Now we close this viewer and see whether its icon is a typescript (see Figure 51). Finally, the last change we made was to use the trafficLight icon for Action Areas. Let's do something that will cause an error, like dividing by zero.

```
&3 ← 1/0
SIGNAL Traps.ZeroDivisor from Traps.ZeroDivisorTrap
computation suspended, switching to Action Area C...
```

Now we'll make the viewer for Action Area C iconic and see if the trafficLight is used.

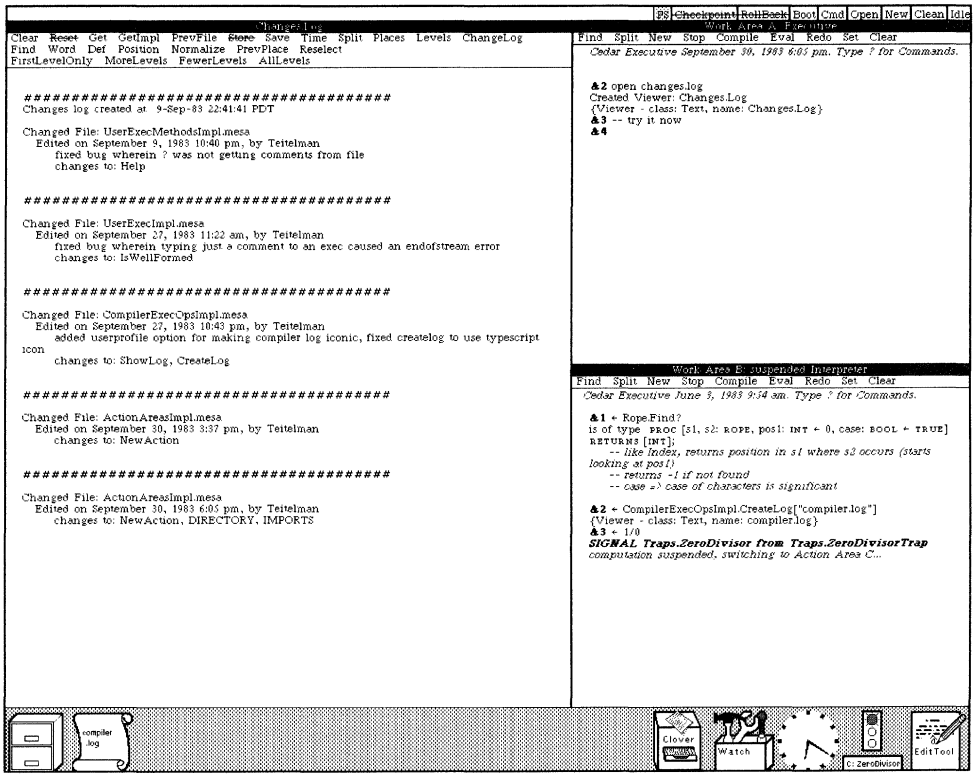


Figure 51

Trying out our changes

Summary

The demonstration contained in this paper has presented a number of the key concepts and facilities in Cedar. Some of these are: a highly visual user interface which exploits the high-bandwidth display and mouse pointing device; a uniform screen paradigm provided by the Viewers Window package, which includes facilities for icons, whiteboards, and tools, as well as text viewers; a high-quality editor and document preparation system (Tioga); spelling correction (DWIM); availability of an interpreter for a compiler-based language; a strongly typed programming language of the Pascal family which also includes automatic storage management, the ability to manipulate types at run-time, and support for Lisp-style lists and atoms; a sophisticated debugger which includes source-object code mapping to facilitate planting of breakpoints and examining program state; support for concurrent operations; and a high degree of integration of facilities and uniformity of user interface. The next paper, "Cedar: The Report Card," evaluates the successes and failures of Cedar.

Cedar: The Report Card

This paper takes a closer look at various aspects of the Cedar programming environment, its successes, its failures, and the lessons to be learned from both. The paper consists of four sections: Catalog Scorecard, Notable Successes, Shortcomings, and What Next? The first section, Catalog Scorecard, grades the current Cedar system against the original catalog of programming environment capabilities drawn up in 1978. The conclusion it draws is that we were successful with respect to most of our high priority items with a few exceptions that are discussed in the section on shortcomings. The second section, Notable Successes, looks in more detail at some areas of Cedar considered (by the author) to have been successes. These include: Object Management, Self Typing Data, Run-time type System, Manipulation of Images, Uniform Screen Management, Remote Procedure Call, Version Control, and Remote File Storage. The third section looks at some of the shortcomings of Cedar in its current form. Most of these shortcomings are in the area of providing support for the Lisp programming style, one of our original goals. This section discusses some of the basic differences between the Lisp and Mesa programming style, and then looks at some specific shortcomings of Cedar including: Fast Turnaround for Small Changes, Support for Wide Range of Binding Times, Easy Use of Programs as Data, and Inheritance/defaulting. The final section, What Next?, discusses some areas that might fruitfully be attacked next.

It is important to bear in mind while reading this paper that while I have attempted to be objective, it is only fair to note that the observations and conclusions presented here belong to the author, and that some members of the Cedar project almost certainly would disagree with many of them.

Catalog Scorecard

Evaluating a programming environment is an extremely difficult task. There is no concept of certification of environments, as there is for compilers, because there is no well-defined notion of what an environment is supposed to do, other than increase productivity. There is no objective metric for the performance of a programming environment, as there is for sort algorithms. Whether or not a programming environment is good or bad depends on what it was trying to achieve, what its goals were. In the case of Cedar, we were in the somewhat unusual position of having articulated our goals before we started ([8] and reproduced in Appendix 1 and 2). Thus, one way of evaluating Cedar is to return to these original goals and grade Cedar on each item.

The original list of capabilities was divided into four categories by priority. The seven priority A items were: object management (garbage collection, reference counting), statically checked type system, memory management (object/page swapping), abstraction mechanisms and the explicit notion of interface, fast turnaround for minor program changes (less than 5 seconds), adequate run-time efficiency, and large virtual address space ($\geq$ 24 bits). With respect to these items, we have done extremely well with one glaring exception: fast turnaround for minor program changes (see discussion under "Shortcomings" below). Mesa already had a statically checked type system, memory management, the explicit notion of an interface, and adequate run-time efficiency. The extension of the virtual address space to 24 bits was a difficult, but straightforward task. The addition of garbage collection to Cedar is discussed in the next section.

There were ten items on our priority B list of capabilities: encapsulation/protection mechanisms (scopes, classes, import/export rules), well-integrated access to large, robust data bases, self-typing data (*a la* Lisp and Smalltalk) and a run-time type system, consistent compilation, version control, source-language debugger, text objects and images, uniform screen management, user access to the machine's capability for packed data, and run-time availability of all information derivable from source program (e.g., names, types, scopes). With respect to these items, we have also attained most of our goals. Mesa already had encapsulation/protection mechanisms, consistent compilation, and user access to the machine's capability for packed data. We successfully implemented a run-time type system, source-language debugger, uniform screen manager (discussed in more detail below), and provided for

run-time availability of all information derivable from source program. Facilities for version control and uniform screen management were provided and are two of Cedar's notable successes (see discussion of successes below). We believe that the Cypress database system [5] is the first step towards providing well-integrated access to large, robust data bases, but the verdict is not yet in.

Successes among the twelve priority C items include: support for interrupts, compiler/interpreter available with low overhead at run time, dynamic measurement facilities, scanned bitmap objects and images, formatted document files, line objects and images, and remote file storage, many of which are discussed in more detail below. Adequate reference documentation is just beginning to appear but work on a librarian and a program-oriented filing system including a browser has not yet begun. Program manipulable representation of programs has not been fully achieved for some of the same reasons that polymorphism and the ability to create fully integrated local sublanguages has failed to materialize (see discussion under "Shortcomings" below).

Other items on our shopping list were checkpoint, menus and other standard user interfaces, document editing, adequate exceptional handling, remote procedure call, message transmission system, all of which have been done. Access to on-line documentation is just beginning to happen. Inheritance/defaulting, the ability to extend the language, and the ability to create fully integrated local sublanguages have not been attacked at all and are discussed further in the section "Shortcomings."

Notable Successes

This section elaborates on some areas of Cedar generally considered to have been successes.

Object Management - Garbage Collection

The addition of garbage collection to Cedar has been an unqualified success; many diehard Mesa programmers that were initially skeptical about the need for a garbage collector in Cedar now state that they would find it extremely difficult to give it up. Although originally thought of as just a convenience for programmers, the addition of garbage collection to Cedar has also caused profound changes both in interface design and programming styles. Without garbage collection, the programmer must insert explicit deallocation statements in all the appropriate places to free storage when it is no longer being used. However, for certain styles of use, it is not always clear that there is any "appropriate place" to insert a deallocation statement. For example, it would be a substantial challenge in a system without garbage collection to ensure that that when no references to a particular stream remain, a backing file would automatically be closed. Thus, the availability of garbage collection to Cedar has enabled a variety of styles of programming not previously available to the Mesa community.

The addition of garbage collection to Cedar required the solution of two problems. The first problem was simply enabling the garbage collector to locate the pointers to objects in collectible storage, since not enough information was present in the actual data itself, i.e., pointers do not carry type information in Mesa. We solved this problem by modifying the compiler to put out additional information in the object code to enable the garbage collector to perform this task.

The second problem was more fundamental: the presence of LOOPHOLEs (breaches of the type system), pointer arithmetic, overlaid variants, and relative pointers in the Mesa language makes it possible for the Mesa programmer to easily fool or confuse the garbage collector without intending to do so. Furthermore, in such a case, a single programming error that smashed a pointer to an object in collectible storage could destroy critical system data structures in ways that would make it difficult to reconstruct, after the fact, any evidence of the original clause of the crash: the system would be, in effect, reduced to a rubble of bits.

In order to resolve this latter problem, we took the approach of identifying a subset of the Mesa language, the so-called *safe* language, that did not contain any of the LOOPHOLE-like features mentioned

above. We then added language constructs to Cedar to draw a protection boundary around a program written in this subset. Even incorrect programs written using this safe subset of Cedar were guaranteed not to be able to interfere with the reliable operation of the garbage collector. The vast majority of Cedar programs are now written primarily (or entirely) in Safe Cedar. (For more information, see [12].)

Self Typing Data, Run-time Type System

The addition of self typing data (REF ANY) to Mesa's statically checked type system was also performed successfully. Self typing data was intended to provide for a form of late binding by allowing the implementor to defer type checking from compile time to run-time on a case by case basis. By employing REF ANY in the early stages of development, programmers could opt for more flexibility at the expensive of performance and/or run-time errors. As the program matured, various binding decisions could be made earlier by employing specific types where appropriate. However, due to the lack of other forms of system support for late binding (see discussion below under "Shortcomings"), this particular use for self typing data did not come into widespread use.

However, we had also envisioned two other important uses for self typing data, both of which were realized and are widespread in Cedar today. The first use is to enable generic programs – programs that can determine the type of a REF ANY at run-time, and operate differently depending on the type of the object they are given. For example, a Sort program could be written which takes as arguments two items of type LIST OF REF ANY, and a comparison procedure of type PROC[x: REF ANY, y: REF ANY] RETURNS[inOrder: BOOLEAN]. The same Sort program could thus be used to sort lists of integers, reals, strings, etc., by supplying a comparison procedure which selected the appropriate metric based on the type of objects being compared.†106

The second use for REF ANY is to enable object-oriented programming (also called closures), where an object in Cedar consists of a block of procedures that defines various operations, along with some private data, represented by a REF ANY, that contains the state information. The use of REF ANY enables the form and representation of the private data to vary between different implementations of the same object, for example, between file streams and keyboard streams. However, from the standpoint of Cedar's type system, these different implementations are nevertheless objects of the same type. Thus, an application can be passed different implementations of a stream without requiring breaching the type system, i.e. a LOOPHOLE. For example, a procedure that takes as an argument an output stream can output material to this stream without caring whether the ultimate destination of the characters is a file, a rope, or a viewer.

In each of the applications discussed so far, the type discrimination being performed at run time is among a collection of types that were known ahead of time, i.e., specified at *compile* time. For example, the generic Sort routine could only sort those objects whose types were built into the comparison procedure, even though that procedure could determine at run-time which of these types was the type of the object it was processing. Being able to perform such type discriminations is sufficient for most applications. However, we also implemented in Cedar a full-blown run-time type system which allows programs to manipulate types as data in a completely general fashion. For example, having determined that the type of an object it has been given is a record, a program can compute the number and names of the fields of the record and obtain the value of the datum stored in any particular field. A Lisp analogy might be to contrast a function that takes an atom and sees whether it is one of a particular set of atoms (very fast) with a function that unpacks the atom into its constitutent characters and then performs some processing on these, e.g., determine if two atoms are the same except for differences in the case of the component characters.

---

†106 In actual practice, if the application will be sorting long lists of such objects or it is important that the sort be as efficient as possible, clients will customize the Sort program to the specific type of the object so as to avoid the run time type discrimination.

Such a capability is admittedly needed only for specialized applications. However, it enables users to implement a class of applications that are normally only in the province of system wizards. For example, the expression interpreter used by the debugger is simply a client of the Cedar run-time type system; it does not include or use any specialized information that is not available to any other client. Another one of the first clients of the run-time type system was a general print routine which took an arbitrary object and printed its value in a manner appropriate to its type. Another interesting application designed by a student is the ViewRec Package, which accepts an aggregate datum (a record, sequence, or array) and constructs a visual interface to this datum. This interface (a viewer) displays and continuously keeps up to date the various fields and values of the datum. It can also be used to change the contents of the data structure.

In most programming environments that are built on top of strongly typed languages, it is simply not possible for an arbitrary user to construct such applications.

Manipulation of Images–Cedar Graphics

There were several items relating to the manipulation of images in our original catalog of programming environment capabilities: text objects and images, line objects and images, scanned (bitmap) objects and images, and formatted document files. This was not surprising, given that the manipulation of images is of primary concern to us in our experimental systems. We divided these manipulations into two categories: manipulation of abstract objects such as formatted documents, forms, line drawings, and continuous-tone images; and manipulation of these objects on displays or printers. Operations in the first category are defined by the semantics of the objects, not by their representation on a particular medium, whereas operations in the second category must take the nature of the medium into account. The first EPE working group believed that "enough experience had been gained in these areas that it [would be] possible to construct packages that will be useful in a wide range of programs, and that will markedly decrease the effort required to write programs that use them" [8] .

The Cedar Graphics package designed and implemented by John Warnock and Doug Wyatt [33] amply confirmed this belief. It provides the support for Cedar's uniform screen manager, the Viewers Window Package, (described earlier in "A Tour Through Cedar" and discussed further in the section below). The flexibility of Cedar Graphics has enabled Viewers to support not only simple typescript-style text applications but also applications involving drawings, scanned images, and combinations of graphics and text. In addition to Viewers, Cedar Graphics has been used to implement a music composition system, a VLSI design system, and a graphic arts design package.

The key idea in Cedar Graphics is a unified graphics *imaging model* and an associated programming interface. The imaging model is totally independent of display devices: it provides an abstraction of how an image would ideally look on a perfect medium. The implementation for a specific device renders the appearance of this ideal image as well as possible. For example, a device that can show grey values might display color values via appropriate grey values, or a binary device might display colors with stipple patterns. Isolating the device-dependent portions of Cedar Graphics into a relatively small set of primitives reduces the cost of implementation for additional devices. For example, Cedar Graphics has already been implemented for binary, grey-scale, and full-color raster display systems, and for high-resolution black-and-white printers and color raster printers.

A good way to think about the Cedar Graphics imaging model is to consider a slide projector shining a general-colored image through a stencil onto a piece of paper, or a silk-screen printer pushing colored ink through a stencil onto paper. The programmer defines sources and stencils via a sequence of procedure calls, and then uses other procedures to produce the effect of pushing a given source through a given stencil. Each stencil and source can also be mapped through any linear transformation prior to display. Very complex images can be built using different combinations of stencils, sources, and mappings. For example, Figure 52 shows the result of using a two-dimensional sampled image of a photograph of our laboratory as a source, and a collection of analytic curves that form the outline of the letters P-A-R-C as

a stencil. The upper image is the result of scaling the same source and stencil by .5 in height and .75 in width.

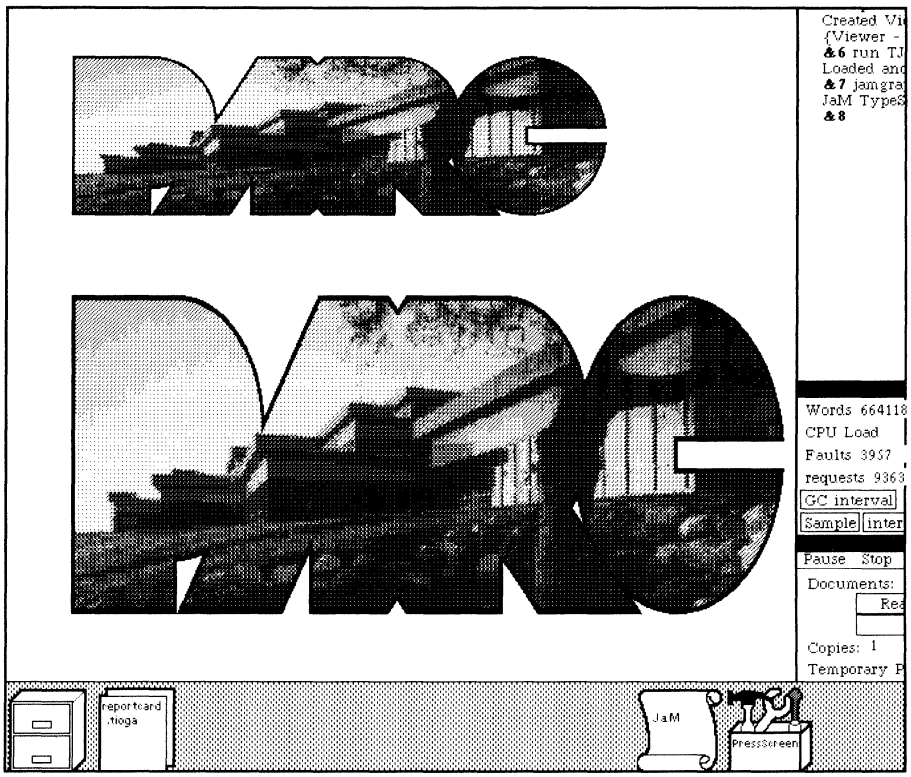


Figure 52

Cedar Graphics enables the display of very complex images

The Cedar Graphics imaging model also includes an additional level of stencil called a *clipping region*, which restricts the area where ink is displayed regardless of what other shapes or masks are used. The Viewers window package makes heavy use of clipping regions. It enables clients of Viewers to ignore the fact that the display is shared among many applications.

Uniform Screen Management–the Viewers Window Package

> Use of the display is pervasive in our interactive systems. Lack of uniformity leads to duplicated effort, often of low quality, since an individual builder cannot easily draw on all past experience or devote the time to taking advantage of it. On the other hand, too much central control over screen management may frustrate the desire to experiment with new paradigms for interaction. We believe that it is possible to "virtualize" the screen and the user input devices–that is, require people to write programs on the assumption that they will only have access to a subpart of the screen, and to a slightly filtered stream of input events–in a way that will not markedly impede our ability to experiment, and that will have a large payoff in terms of the user's ability to construct a screen environment containing multiple windows on different programs. [8]

The Viewers Window Package has successfully attained these goals. It is "the arbiter of the user input and display hardware in the Cedar programming environment. It provides the illusion to the programmer that there is a private display, mouse and keyboard associated with each application, while

allowing the user to simultaneously interact with many such applications" [19].

Resolving this issue of distributing user actions among various applications all running concurrently was one of the most interesting and challenging issues in the design of Viewers. The problem is that while some user actions, such as mouse clicks, include information that determines the application to which they are directed, namely the location of the cursor at the time the mouse is clicked, many user actions such as keystrokes do not contain such information. Instead, the recipient of these latter actions is a function of previous user or program actions.

This problem is handled in Viewers by defining the notion of an *input focus* that associates keyboard activity with a particular viewer. The corresponding application program reads characters from the keyboard using the standard pull model: the program asks for a character, and when characters are not available, the program waits. Note that characters may not be available to this program because the user has not typed any, or because the user has redirected his typing to some other application.†107 In fact, one of the virtues of this scheme is that these two cases are indistinguishable to the program. In other words, the program can ignore the fact that the keyboard is shared, just as the clipping region in Cedar Graphics makes it possible for programs to ignore the fact that the display is shared.

Another issue successfully addressed by Viewers was the desire to facilitate experimentation with various user interfaces. The Terminal Interface Package (TIP) makes it easy for individual applications each to employ different user interfaces, and for individual users to change existing interfaces to meet their own preferences. TIP achieves this by separating *interface* from *function*, i.e., separating the operations implemented by a particular application, such as Delete, Exchange, and SelectNode for Tioga, and Select and Draw for the IconEditor, from the way in which the user invokes the operation, such as typing CTRL-X, or double-clicking the left mouse button. A specially formatted file called a TIP table specifies the mapping of user actions into system operations. For each application, TIP parses the user actions into the corresponding operations using the application's own TIP table. This arrangement makes it easy for individuals to change or extend the user interface. Figure 53 shows a portion of the TIP table that interprets user actions inside of icons. For example, the line marked with an asterisk in this table specifies that when the LeftShift key is down and the user clicks the middle button on the mouse, the icon that the mouse is in is opened and given the full column.

The Viewers Window Package allows programmers to create new classes of viewers by specifying an implementation for various operations, such as how to display a viewer that is an instance of this class, what cursor shape to display when the user moves the mouse into this viewer, what to do when the user clicks the mouse in the viewer, how to scroll the viewer, etc. This facility has been used for implementing a wide variety of viewers. These include whiteboards (Figures 5, 6), sliders (used for displaying and setting continuous values), histograms, graph browsers, record viewers, plus various tools such as the Watch tool (Figure 15), File tool (Figures 2-4), EditTool, TypeSetter, Clock (Figure 2), and games such as MazeWar, Tank, and Football (see Figure 54).

One shortfall in the current implementation is lack of support for subclassing; it is not possible to define a new viewer class by specifying only the ways in which it differs from some existing class. For example, it would have been very useful if Walnut, the electronic mail reader, could have defined a class of viewers called Message Senders, which were like Tioga viewers in every way except that, in addition, they supported the operations of Send. Although subclassing in general is not supported in Cedar (the

†107 The user changes the input focus from one viewer to another simply by clicking the mouse in the corresponding viewer. Applications can also change the input focus. For example, when the user clicks Walnut's NewForm menu button, a message sender viewer is created, and the input focus placed in that viewer. Similarly, when a breakpoint is encountered and a new Action Area created (see Figure 24 in "A Tour Through Cedar"), the input focus is automatically placed in the Action Area.

lack of such support is discussed separately below), it would have been possible to provide some form of support for subclassing for Viewers.†[108] This issue has recently begun to receive some attention.

```
SELECT TRIGGER FROM
    Left Down      => Select;
    Left Up WHILE Ctrl Down    => Delete;
    Middle Up      => SELECT ENABLE FROM
        Ctrl Down  => OpenDesktop;
        LeftShift Down    => CloseOthers, Open;
        ENDCASE           => Open;
    DEL Down  => Delete;
    L Down    => Left;
    M Down    => TogglePos;
    O Down    => SELECT ENABLE FROM
        Ctrl Down  => OpenDesktop;
        LeftShift Down    => CloseOthers, Open, SetInputFocus;
        RightShift Down   => CloseOthers, Open, SetInputFocus;
        ENDCASE           => Open, SetInputFocus;
    R Down WHILE Ctrl Down    => ResetDesktop;
    R Down    => Right;
    S Down    => Save;
    ENDCASE.
```

Figure 53

The Terminal Interface Package facilitates experimenting with user interfaces

Another shortfall of Viewers is that it is difficult to build complex windows with much internal structure such as that employed by the Watch tool (see Figure 15). The principal reason for this is that the language by which the client constructs a viewer is *imperative* rather than *declarative*: the client specifies the algorithm for laying out the display via a sequence of procedure calls which perform the layout. As a result, the various decisions about the display are wired into a program, and hence difficult to change. Furthermore, it is difficult for all but an expert to read the program and visualize the resulting display. Specifying such a window via a passive data structure which described the desired result and was interpreted at run-time might provide a solution to both of these objections. Another approach would be to design a tool for constructing such viewers, which would allow the user to experiment interactively with various layouts, inserting, deleting and moving the components of the viewer being constructed. When a satisfactory result was achieved, the corresponding program or data structure could be constructed automatically. (Such a tool has been implemented in another programming environment at Xerox and has been enthusiastically received.) A more general solution to this problem would be to allow Tioga documents to include viewers and other graphical entities along with text. In this case, the Watch tool would simply be a Tioga document, and could be created, edited, and saved accordingly. Future plans for Tioga call for such generalized documents.

†108 In fact, an earlier implementation of a screen manager for Cedar did include support for subclassing.

Remote Procedure Call

The ability to call a procedure on another machine as though it were on one's local machine was one of the package items in our catalog of capabilities. For those unfamiliar with this notion:

> The idea of remote procedure calls (RPC) is quite simple. It is based on the observation that procedure calls are a well-known and well-understood mechanism for transfer of control and data within a program running on a single computer. Therefore, it is proposed that this same mechanism be extended to provide for transfer of control and data across a communication network. When a remote procedure is invoked, the calling environment is suspended, the parameters are passed across the network to the environment where the procedure is to execute, and the desired procedure is executed there. When the procedure finishes and produces its results, the results are passed back to the calling environment, where execution resumes as if returning from simple single-machine call. [2]

The primary purpose of RPC in Cedar was to make distributed computation easy. We had observed that building communicating programs in our research community was a difficult task, one which was attempted only by a select group of communication experts. Even researchers with experience in building systems found it difficult to build distributed systems with our existing tools. We considered this state of affairs to be very undesirable:

> We have available to us a very large, very powerful communication network, numerous powerful computers, and an environment that makes building programs relatively easy. The existing communication mechanisms appeared to be a major factor constraining further development of distributed computing. Our hope is that by providing communication with almost as much ease as local procedure calls, people will be encouraged to build and experiment with distributed applications. [2]

We are still in the early stages of acquiring experience with the use of RPC and certainly more work needs to be done, but it appears that Cedar RPC has achieved its goals of making distributed computation easy. There have already been several projects that have used RPC to implement various distributed applications. These include the control communication for an Ethernet-based telephone and audio project and the complete communication protocol for Alpine, a file server supporting multi-machine transactions and page-level access. Here is a comment from one of the implementors of Alpine:

> Using RPC has proven valuable for the following two main reasons: 1. It frees both client and implementor from worrying about the actual format of the bits going over the wire. An ordinary Mesa interface module is the total description of the arguments and results of all operations performed by the server. Furthermore, the RPC semantics are sufficiently complete (i.e., similar to single-machine procedure call) that the programmer doesn't have to think very much about adapting an interface for remote use. 2. RPC takes care of all aspects of remote binding, authentication, and reliable communication. *One does not need to be a communication wizard in order to communicate.* [italics mine]

Several network games have also been implemented using RPC. The basic paradigm for each of these programs is that there is one server to arbitrate among all of the players. This server acts as a clearinghouse for the state of the game. Each player calls in with his state and gets back the state of the world. Figure 54 shows a two player football game implemented using this scheme.

Such applications fall under the category of 'closet' projects: they would never be attempted if they looked like large tasks. Here is a testimony from one of the game implementors: "How easy was it to use? Astoundingly! The first 90% came in a couple of hours one afternoon (the last 10% dragged on at least in part because of irrelevant reasons). Did it help me? I think it made the difference between this being a small toy and being a big project."

Another testimonial to the success of RPC in Cedar is that as a result of our experiences, the RPC protocol has been implemented in both Interlisp and Smalltalk, thereby allowing applications running in

entirely different environments to communicate with one another.

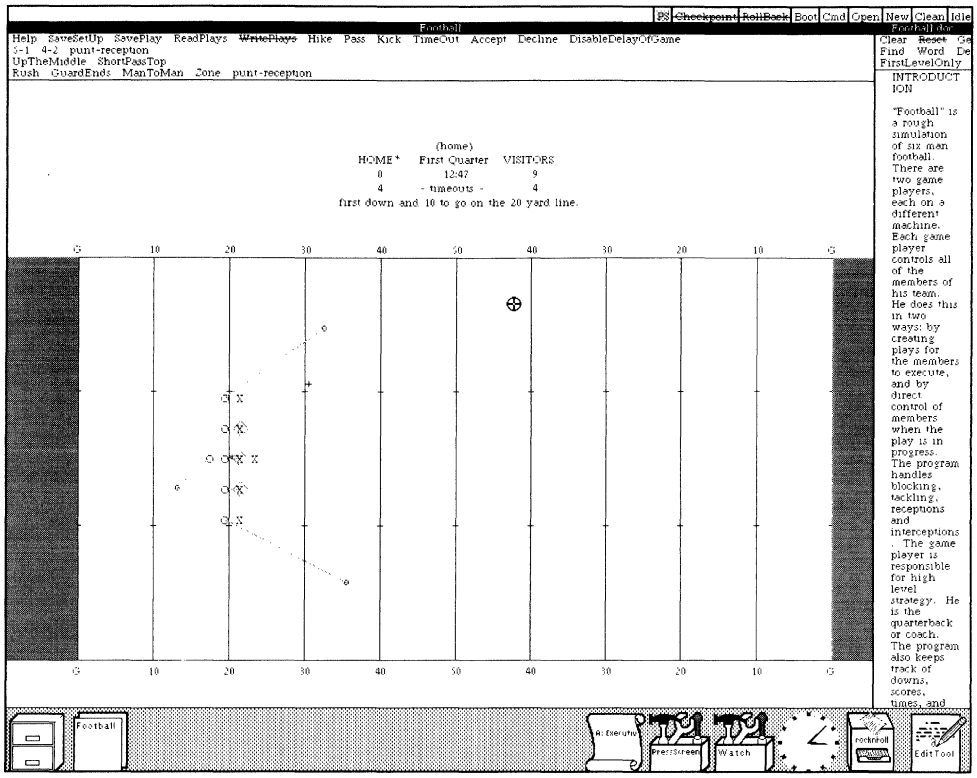


Figure 54

RPC enables implementing network applications and games

Version Control

Cedar programmers work in a distributed computing environment, and have to be able to share each other's programs in various stages of development. In this setting, control of versions and file management is difficult†109 both because of the large number of files in Cedar and the requirement that versions of files must agree.†110 In anticipation of these factors, consistent compilation and version control was one of the priority items in the catalog of programming capabilities.

The principal facility provided by Cedar for dealing with version control is *description files* (DF files). DF files contain information about versions of files needed by a particular application and their locations. There are several packages available for manipulating DF Files, of which the most frequently used are: BringOver, for retrieving all or some subset of the files specified in a DF File; SModel, for storing files on a server; and VerifyDF, for checking that the files mentioned in a particular DF File are all consistent, i.e., that if two files import a common interface, they both use the same version of that interface.

†109 In fact, the problem of controlling software development in a distributed environment turned out to be sufficiently interesting and difficult that it yielded a PhD thesis for one of the Cedar implementors, Eric Schmidt.

†110 Mesa ensures consistent compilation by placing time stamps on source and object files, and by recording in each object file the complete list of time stamps for the files that produced it.

Based on our experiences with the DF System, we attempted the much more ambitious task of providing a complete program management system:

> The user describes his software in *system models*, which are complete descriptions of a software system. Similar to a blueprint or schematic, a model combines in one place 1) information about the version of files needed and hints about their locations, 2) additional information needed to compile the system, and 3) information about interconnections between modules, such as which procedures are used and where they are defined. System models are manipulated by the *System Modeler*, a program that automates development of software in the Cedar programming environment. The system modeler is notified of new versions of files as they are created by the editor, and automatically recompiles and loads new versions of software. [26]

The system modeler has only been used by two or three users, so the verdict is not yet in on its utility. However, it is clear that the DF system was a major success in Cedar. Not only did it automate version control for system implementors, but users soon found DF files indispensable for managing their own private software. This was especially true for users working on public Dorados–machines that were shared among several users. Even for those users fortunate enough to have their own private machine, the maxim of "Keep your bags packed" turned out to be good advice in an experimental and rapidly evolving system such as Cedar, and DF files enabled the user to do just that.

Remote File Storage

The original catalog of programming facilities included as a package item Remote File Storage, i.e., automating the transfer of files between machines: "The manual transfer of files between machines is a significant source of errors and wasted time. Such transfers are necessary either because of space problems, or because one machine has a capability (such as a printer or high-performance display) not possessed by all" [8].

The DF system described above took a large step towards eliminating the manual transfer of files, or at least the errors and wasted time associated with this activity. By executing a single BringOver operation, the user could reestablish a consistent set of files on a particular machine.†111

However, we did not attain the goal of unifying the local and remote file system into a single uniform, shared file system, so that the user need never be aware of where files are stored and could simply treat his local disk as a form of temporary memory. Instead, the recently designed and implemented Cedar File System (FS) represents a compromise position between such a uniform, shared file system and what we had before – completely independent local and remote file systems with manual transfer of files – while it eliminates the need for manual transfer of remote files, it does not completely mask the existence of the local disk. In some cases the user must be aware of the distinction between files that exist in the local file system. For files that exist on the local file system and nowhere else, the user is responsible for storing the local file on a remote server.

The reason for this compromise position was that we wanted to preserve the idea of a strictly local file, because users were familiar with it and found it useful. Another reason for not unifying the local and remote file system was that we were committed (at least initially) to using an existing file server which did not provide support for transactions; the absence of transactions makes it hard to synchronize shared file access.

---

†111 For those files that are part of the system such as sources and fonts, the user need not be aware of their location, or even their existence: Cedar automatically retrieves them when needed without requiring any action by the user.

It is not clear whether we will eventually design and implement a unified file system. It is also not clear, given that the FS is only just beginning to be used, how effectively it will allow users to operate in a mode in which they do not worry about where files are stored. Success or failure of the latter will undoubtedly influence the former.

Shortcomings

Building a programming environment is an extremely difficult task. There are no solutions, no right or wrong, just choices. Furthermore, the task is unbounded: you never finish a programming environment, you simply gradually stop working on it (usually about the time you start planning another). One of the most difficult challenges that faced us in the Cedar project was to decide what we would include. There was considerable divergence on this subject among individuals in the project. This section reports on some of the things that we chose *not* to do, why we made these choices, and how they have affected the overall result. It is perhaps the most subjective part of the paper, and some members of the Cedar project might disagree strongly with my conclusions. However, being the only Cedar implementor from the Interlisp community, I was in the unique position of being proficient and intimately acquainted with both Mesa and Interlisp. This perspective that has emboldened me to set down my observations and conclusions in the hopes that they might be of interest or value to designers of future programming environments.

The principal shortcomings of Cedar are in the area of providing support for various aspects of the Lisp style of programming (and to a lesser extent Smalltalk), and can be attributed to the selection of Mesa as a starting point for Cedar and the fact that the overwhelming majority of the Cedar implementors and users came from the Mesa community. These shortcomings include not reaching Cedar's original goals with respect to: fast turnaround for small program changes, support for wide range of (i.e., late) binding times, easy use of programs as data, and inheritance/defaulting (Smalltalk subclassing). In short, with respect to the fundamental principle stated in the EPE report [8] that "the present Lisp, Mesa, and Smalltalk programming styles all must be supported in a satisfactory way," it is fair to say that Cedar has not (yet) succeeded.

However, it should be pointed out that while Cedar has not succeeded in these areas with respect to the original goals as stated in 1978, these goals themselves were revised and modified as the Cedar project developed. For example, in 1980, Jim Morris, then manager of the Cedar project, stated:

> Acceptance of this specific goal of increasing programmer productivity, and its immediacy, i.e., over the next several years, has made us conservative in our designs. In the main, we have restricted ourselves to those ideas which can be understood and put to use by the intended users in a timely fashion. While it is our business as Computer Science researchers to strive for new and revolutionary ideas, they are not required for Cedar. Indeed, employing the users of Cedar as guinea pigs for such ideas would tend to decrease their programming productivity in the time frame of interest. [21]

Thus, in 1980 we were already beginning to recognize that we might have been overly ambitious, and to reduce our aspirations. By 1982, the Cedar project was being described as "an attempt to take the Mesa language and build around it a programming environment based on ideas from Interlisp and Smalltalk, while retaining the strong type-checking properties of Mesa" [26]. In other words, by 1982 the goal of building an environment that would be attractive to both Mesa *and* Lisp users had been discarded.

It is not the author's intent to cast a value judgment about how Cedar has developed: in the face of limited resources, choices must always be made about which areas to attack first. It is only natural and proper that such choices be made in terms of the greatest good for the greatest number, and the greatest number of users (and implementors) of Cedar came from the Mesa community. From the standpoint of these users, Cedar has been an unqualified success: they are overjoyed at the increase to their productivity that Cedar has provided them when compared with the previous Mesa programming

environment. The fact that we did not achieve certain goals should not be taken as an indication that these goals are not attainable in an environment based on a strongly typed language. Some of these goals are not even technically difficult compared with some of the things that we did accomplish. We made some choices, and this section reports on these choices and their consequences. In the future, we may in fact revisit some of these choices – there is still the possibility that as Cedar matures we will address some of the shortfalls discussed here.

Since the shortcomings listed above all relate to lack of support for various aspects of the Lisp programming style, before we examine each of these specific areas in detail, it is appropriate to discuss the basic differences between the Lisp programming style and that of Mesa. Such a discussion will help to explain why the Mesa, and hence Cedar, user community placed less importance on various issues that are considered absolutely essential to the Lisp community.†112

A Matter of Style – Lisp versus Mesa

The principal differences between the Lisp and Mesa style arise from the types and purposes of the programs they write:

> Lisp is used almost entirely as a research tool. ... The average Lisp user writes a program as a programming experiment, i.e., in order to develop the understanding of some task, rather than in expectation of production use of the program. The act of developing the program, not the act of running it (even for test data), constitutes the experiment. As a consequence, the program is likely to be large and complex, to undergo drastic revisions while it is being developed, and to be thrown away before it has been "completed" by conventional programming standards since it will already have served its purpose before then. [25] †113

Beau Sheil [27] has called this style of use:

> *exploratory programming*, the conscious intertwining of system design and implementation. ... 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 the design and program develop together.

Lisp evolved in response to the need for programming environments that facilitated this exploratory style of use. For example, Lisp systems were first developed to support research in artificial intelligence, where the programmer "invariably has to restructure his program many times before it becomes reasonably proficient. In addition, since intelligent activities are complex, programs tend to be very large, yet they are invariably built by very small teams, (often a single researcher)" [27].

Mesa, on the other hand, evolved in response to a need for producing reliable, robust systems, developed by large teams of programmers, and the ability to maintain such systems over a fairly long period, often by programmers who were not the original implementors. For example, the mandatory, compile-time type checking employed by Mesa is particularly useful in the maintenance of large programs: the additional, redundant information contained in the type declarations makes Mesa programs more readable by others.†114 The type checking also gives greater confidence that when changes are made to programs, trivial new bugs will not be introduced.

---

†112 For additional discussion, see the section entitled "Character of the Result," in "The Roots of Cedar," the first paper in this report.

†113 Admittedly, this situation has begun changing in recent years. Increasingly, Lisp systems are being used to implement reliable programs intended for production use. However, the thrust of the comments here concerning the difference in style between the two communities is still valid.

†114 After much experience with both Mesa and Lisp, in the author's opinion, it is a lot easier to write and get working your own program written in Lisp, but much easier to read or modify someone else's program when it has been written in Mesa.

The Lisp programmer would argue that the advantages provided by type-checking are not significant for the kinds of programs that he typically writes:

> The advantages will be small for programs whose "characteristic times" (design, programming, checkout, existence, total execution) are all measured in minutes, large if they are measured in weeks or months. In an environment where programs are undergoing rapid change, [Mesa's] mandatory checking mechanisms tend to introduce unnecessary overhead by requiring complete internal consistency at every step of the development process. [11]

In fact, the requirement that the types of all values in Mesa must be specified in advance is considered by most Lisp programmers to be a nuisance and an irritant, rather than an attractive feature of the Mesa language. Here is a typical comment: "I think that static type-checking is a waste of time; it solves a small number of problems while creating many more. I and other Lisp programmers spend a very small percentage of time chasing problems static type-checking would catch."

Because of the need for type declarations and specification of interfaces, Mesa requires more planning before a running program is created than does Lisp (some would consider this a disadvantage, others a benefit). Similarly, the Mesa programmer tends to put more thought and planning into each change.†115 In compensation, the Mesa programmer is fairly confident that once his program is finished and has compiled successfully, he will spend much less time debugging it.†116 This is extremely important to the Mesa user because the process of finding and fixing bugs in Mesa programs is much more painful and time-consuming than it is for his Lisp counterpart, as is discussed in more detail in the next section.

A definite weakness of the Mesa approach is that it is relatively difficult to add flexibility that was not anticipated in the original design. Furthermore, Cedar programmers rarely anticipate and provide for generalizations ahead of time – before a particular situation is encountered requiring them – because of a viewpoint that is more or less prevalent throughout the Mesa community, and perhaps best summed up in [16]: "An interface should capture the minimum essentials of an abstraction. Don't generalize, generalizations are generally wrong." This philosophy has also been stated as "If in doubt, leave it out," and "KISS: Keep it Simple, Stupid." As a result of this attitude, with which the author disagrees, there is often a significant time delay in Cedar between a perceived need and a capability which meets this need. This is detrimental: it hinders our ability to experiment.

The Role of Change in Program Development

Perhaps the area of greatest difference between the Lisp and Mesa communities is in how each views the process of change. The Lisp programmer tends to view change as an integral and *desirable* part of the program development cycle. A typical Lisp debugging session has a "stream of consciousness"

---

†115 This philosophy of "go slowly, don't make mistakes because they are expensive to correct" seems to carry over into the way the Mesa/Cedar user interacts with the system, which is at a more deliberate, somewhat slower pace than that of their Lisp counterparts. It is perhaps for this reason that DWIM, the automatic error-correction facility, did not receive as widespread acceptance in Cedar as in Interlisp.

†116 This is an extremely fuzzy area. It is true that the time a Mesa programmer spends debugging his program is often much less than the time spent debugging the corresponding Lisp program. For example, my very first programming effort in Mesa, a spelling corrector, took three days to get to compile, a process which I found extremely frustrating (and was undoubtedly aggravated by my inexperience with the Mesa syntax). However, once compiled, my program was debugged and running in half an hour.

However, these times can be misleading because the Lisp programmer starts debugging much earlier in the program development cycle, i.e., at a point where the Mesa programmer is still designing his algorithms and data structures. Debugging and design are often intermixed in the Lisp style. In one experiment which compared the overall time from start to finish for a programming problem (reading in a text file and performing simple justification), the Lisp programmers in our laboratory did much better than Mesa programmers. However, this experiment involved a program whose characteristic time was quite small, and has already been pointed out, Mesa's benefits come with larger, longer-lived programs.

flavor to it, rather than the deliberate, planned attack that a Mesa programmer is more likely to adopt. The Lisp programmer simply starts using his program, and analyzes and fixes problems as they come up. While pursuing the first problem he encounters, the Lisp user will often encounter a second, which leads to a third, and so on. When this happens, the Lisp user frequently, to use a programming metaphor, pushes the original problem onto his stack, and pursues the new one. The facilities of the Lisp system supports this paradigm, and also provides tools for the programmer to keep track of what he is doing.

The Mesa system, especially the multiple threads of control provided by the process mechanism, also enables the programmer to suspend a particular debugging path and pursue a new problem that he has just encountered, or conversely, to continue pursuing his original problem, while leaving suspended the new problem to which he can return later. The key difference between the two systems is that when the Lisp programmer analyzes a problem, he (usually) can fix it on the spot and *continue his debugging session with the fix now in place.*†117 A single Lisp debugging session may last several hours during which time the programmer will find and fix a number of problems. The interactive nature of this process is especially important given the kind of programs a typical Lisp programmer often writes, where the problem being solved, much less the algorithms being used, are not well understood, hence the need to "debug the program into existence" by experimenting with various solutions and seeing how they work.

Lisp systems have been used in this highly interactive fashion for more than a decade. Over that period, considerable effort has been devoted to building tools which facilitate this style, especially with respect to making changes and continuing the debugging session. For example, the debugger and editor are integrated to allow the user, having identified a particular place on the call stack, to edit the corresponding expression in the source. (Cedar has a similar facility.) When a problem is not detected until after the damage has been done, the user can alter the flow of control from the debugger, returning the computation to a specified place on the call stack from which he can then continue with the fix in place. There is even a facility, the Advise package, which allows the user to experiment with the effects of a proposed change without having to perform any edits. Advise operates by redefining Lisp functions so that the indicated expressions are evaluated at the entry or exit of a procedure. Advise can also operate on a specific call to that procedure, such as the call to Print from within the function Foo. Finally, the Interlisp file package keeps track of the changes that the user makes to various program elements, and informs the user which files need to be saved.†118 The file package also notes changes to elements which are not associated with any particular file, such as is the case when the user defines new functions during the course of a debugging session. All of these facilities allow and encourage the user to find and fix many bugs in a single session, building up and retaining as much state and context as he wishes during the process. This paradigm seems much more effective than having to break the debugging process up into a sequence of small sessions consisting of find some bugs, fix the bugs, start over.

It is difficult to distinguish cause and effect in the evolution of the Lisp style: did the tools develop in support of the exploratory style, or did the existence of these tools encourage the growth of the Lisp style? (Probably the former is the case.) Similarly, it is difficult to separate cause and effect in the relative lack of support for making changes in the Mesa environment. Historically, making changes was always extremely costly in terms of programmer time in the Mesa environment. When a change was required in a Mesa program, the programmer had to edit his source,†119 compile it, correct syntactic errors (except

†117 Note that this is not a compiler versus interpreter issue, but one of dynamic relinking. Many times, especially where performance is an issue, the Lisp user will take the time to compile his changes, although Lisp does not require it, i.e., interpreted and compiled code can be freely intermixed. The important point is that regardless of whether it has been compiled or is being executed interpretively, the modified version will be the one that is executed for all subsequent calls to the program.

†118 the Interlisp editor operates on the loaded, structure representation of programs, rather than on source files.

†119 In the Alto Mesa world, editing required leaving the debugging environment and running an entirely separate editing system. This made making changes even more painful than it currently is in Cedar where the editor and debugger are integrated into a single environment.

for minor changes, it is unusual for a program to compile successfully on the first attempt), recompile, perhaps several times, and then *abandon his current context and start anew* in order to load the now changed program before he could evaluate the effects of his change.†120 The absence of an interpreter (something that has been corrected with Cedar) meant that the programmer also had to construct and debug test programs and data structures for exercising his program.†121 If the programmer is developing a multi-module system which includes various interfaces for communicating between its parts, a simple change to one of these interfaces might require recompiling the entire system. (See the discussion contained in "Recompiling Interfaces" below.)

Because making changes was so hard, they were avoided as much as possible. Considerable emphasis came to be placed in the Mesa community on "getting it right the first time." For some, it became a matter of pride: the Mesa programmer often views the need for a program change as an indictment of the original design or implementation, an indication that something was done *wrong*.†122 Thus, providing facilities that facilitated change tended to be given lower priority than other environmental issues.

Fast Turnaround for Small Changes

The key ingredient in the Lisp style discussed above is fast turnaround for small changes: the Lisp programmer makes a change and sees the change take effect immediately. When we began work on Cedar, the turnaround time for a Mesa program change was often measured in terms of dozens of minutes. This time lag forced the programmer to operate in a fashion that resembled batch processing, even though he was operating on a personal, dedicated machine. The programmer would identify as many problems as possible in a single debugging session, then go off and make the required edits that he hoped would fix these problems, and then resubmit his job and see if the changes worked. This resulted in a tremendous loss in productivity as compared with a programmer performing a similar task in Lisp or Smalltalk.†123

---

†120 For a certain, not terribly well-defined class of programs, it was possible to load multiple instances of a program into the same environment, i.e., on top of one another. However, this was not a practice that was encouraged or widespread because of the possibility of confusion as to which instance a particular client was bound. For example, if A calls program B, and B is changed and reloaded, program A continues to be bound to the original version of program B. On the other hand, programs loaded subsequently, such as a newer version of A, will be bound to the latest version of B.

†121 Earlier Mesa environments had an interpreter for a (not well-defined) subset of the Mesa language. However, for the purposes of experimentation, this interpreter had two serious shortcomings. First, because the debugger did not share the same address space as the client, it was not possible to perform operations involving storage allocation. For example, the user could call a procedure on the value of a datum that existed in his current computing context, but he could not construct such a value on the fly to supply as the argument to a procedure. Second, it was not possible to save and reuse the values of expressions given to the interpreter, e.g. assign the value of an expression to a newly created variable. Thus, the user could not decompose interpreting a complicated expression into several simpler operations. The Cedar history facility, combined with the residential nature of the debugger and interpreter, has successfully resolved both of these issues.

†122 An extreme version of this negative point of view regarding making changes easy was presented to me by an Air Force Colonel at a programming environment workshop. We were having a discussion about the merits, and drawbacks of the automatic spelling correction facility in Interlisp (DWIM). He was concerned about the possibility of DWIM making an inappropriate correction to a program. I assured him that the user was always informed, that corrections had to be confirmed, and that they were easily undoable. He remained unconvinced. I then proposed that we eliminate spelling corrections to programs, and consider only corrections to the instructions that the user gives the operating system, such as load this file, run that program, etc. I maintained that correcting such mistakes improved productivity. His position was: "When one of my programmers makes a mistake, I don't want the system to help him out. I *want* him to have to go home and think about it overnight." This point of view is not confined to the military. A leading European spokesman for modern programming technology is on record as having stated that programs should not have to be debugged, and that the only programming tools a good programmer should need are pencil and paper.

†123 Many Mesa programmers were not aware of this loss of productivity – they had never had the opportunity to develop their programs in a truly interactive fashion.

In Cedar, we were concerned with rectifying this situation and providing fast turnaround for small program changes. "Our concern with fast turnaround comes from the observation that programming should be *think bound*, not *compute bound*." Mesa offered several medium-size obstacles to fast turnaround for changes: the editor was not integrated or even properly packaged, the compiler was not designed to compile anything smaller than an entire module, and the system did not provide incremental replacement of procedures or even modules. [8]

The development of the Tioga editor within the Cedar environment overcame the first of these obstacles, but we never did mount a serious effort to attack either of the latter two issues: compiling individual procedures and replacing modules. Both of these problems were much harder than we originally anticipated, partly because of the monolithic nature of the compiler, and the difficulty of changing or reorganizing it significantly.

Instead, the goal of fast turn around for small changes was transmuted to the goal of reducing the overall time spent in the edit-compile-reload cycle, i.e., speaking metaphorically, giving the batch programmer faster turnaround, rather than providing him with interactive access to the machine. Since the editor was resident in the Cedar environment, it was no longer necessary to abandon program state while making changes. Since the user was free to perform other tasks such as reading mail, editing, or even debugging other parts of his program while waiting for a compilation to finish, the edit-compile cycle became significantly less painful. Furthermore, the availability of a checkpoint-rollback facility (in Interlisp parlance, Sysout and Sysin) reduced to approximately 30 seconds the time required to return the system to a pristine state into which a changed version of a program could be loaded. Thus, the entire edit-compile-reload cycle was reduced to on the order of a very few minutes.†124 Nevertheless, the situation was still qualitatively very different from that of the Lisp programmer who could make a change and see it take effect immediately.

Note that the key ingredient here is not necessarily the time required to see a change take effect, but whether the change can be made *in situ*. Even if restarting the entire system and reloading it with changed programs could be performed *instantaneously*, Cedar would still not have achieved its goal of providing fast turnaround for small program changes in the sense that it was originally conceived and is provided by Lisp and Smalltalk. There would still be a need to replace an instance of a running program with a changed version of the same program to *preserve valuable program state*. For example, over the course of a lengthy session, the user may have built up a complicated data structure and program state in which he wants to test out the effects of a proposed change. In such cases, it is desirable to replace running programs with changed versions, even if this operation takes *longer* than restarting the system and reloading the changed program. Cedar has failed to provide this capability. In the author's opinion, this is the single biggest shortfall of the Cedar project.

†124 To take the example of fixing the off by one error in the file UserExecMethodImpl as shown in Figure 27-28 of "A Tour Through Cedar," the time required to make the edit itself is 1-3 seconds, to save the file is approximately 10 seconds, to compile the file another 25 seconds. (Both of these times obviously will depend on the size of the file. The file in question is larger than average by Cedar standards, about 20,000 bytes.) If this were the only change we were going to make and we wanted to test it out, we would then have to rebind the configuration which includes this file,. This takes another 20 seconds. Then we would rollback – another 30 seconds – and finally run the new configuration, which in the case of the userexecutive takes about 20 seconds.

One final comment on the subject of changes: while Cedar is weak in the area of making changes whose implementation should take on the order of minutes, it is very strong, much stronger than Lisp or Smalltalk, in the area of making changes which normally take on the order of days or weeks, such as drastic reorganizations of basic data structures or algorithms in a large system. This is because the explicit notion of interfaces in Cedar, combined with the enforced type checking, serves to detect right away most of the problems that would only surface over a period of time if the corresponding changes were made in Lisp or Smalltalk. Furthermore, once the changed system has been successfully compiled and loaded, the programmer is fairly confident that it will run, whereas the Lisp or Smalltalk programmer must embark on a lengthy checkout operation to make sure all of the things that used to work still do.

Support for Wide Range of Binding Times

According to Beau Sheil [27]:

> 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 interest of minimizing and deferring the cost of making large-scale program changes. The languages make extensive use of *late binding*, i.e., allowing the programmer to defer commitments as long as possible. ... [One example of late binding is] 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 structures 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.

By contrast, "the Mesa style requires relatively early binding of many aspects of programs that in Lisp are typically bound during execution" [11]. Thus, it is relatively difficult to add flexibility that was not anticipated in the program design, thereby restricting the range of experiments that can be performed easily. We were aware of this problem in our early discussions. We agreed that:

> The EPE must support a wide range of binding times, including the Mesa and Smalltalk extremes, in a way that allows changes in binding time without structural changes in the program. Different choices of binding time by the programmer may lead to different turnaround times for apparently minor changes, and to different execution efficiencies, but the *functional* behavior of programs must not depend on such choices. [11]

We intended that Cedar make provision for binding at a variety of times, but that delayed or dynamic bindings would occur at the programmer's request, rather than by default as is the case in Lisp. This would allow programmers accustomed to the Mesa style to continue operating in the manner with which they were familiar. For those wishing to adopt the Lisp style of delayed binding, tools would be available to exploit program redundancy to infer suitable declarations for programs written without them, and otherwise make it easy to convert programs originally written in a delayed-binding style to earlier bindings for greater efficiency and ease of maintenance. For example, one reason for the addition of REF ANY to Cedar's type system was to allow the programmer to defer type checking from compile time to run-time. At some later point when the program stabilized, one could imagine a tool which would assist the programmer in the task of converting these REF ANY declarations to specific types where appropriate. However, we never did get around to building such tools, and as a result, it is unusual for programmers to use late binding, and then convert to earlier binding at some later point after the program matured. However, as discussed earlier, many programs do use REF ANY for other purposes.

Easy Use of Programs as Data

The keystone of the Interlisp programming environment is its very large repertoire of facilities that support the user in the task of program development. These facilities include a sophisticated structure editor, a history package that provides both a Redo and Undo capability, automatic error correction, the Advise package discussed earlier, a package for analyzing user programs to determine various calling and usage relationships, and others. Underlying most of these facilities is the easy use of programs as data, i.e., a convenient, program-manipulable representation of programs.†125 In fact, in the author's opinion, the equivalence of programs and data in Lisp, i.e., the fact that Lisp programs are simply list structures, is the *single most important aspect of Lisp.*†126

We wanted to see a collection of facilities comparable to those in Interlisp developed in and for Cedar. We also wanted to make it possible for the average user to contribute to this collection of tools: "Perhaps the single most important observation about the use of Lisp is that as users become more experienced, they start building tools within the system to help them" [11]. Therefore, the issue of program-data equivalence received special attention in our early discussions. In particular, we identified three major facilities within the Mesa environment that would be necessary to enable treating Mesa programs as data: Lisp-style atoms, universal pointers (pointers that carry the type of their referent with them), and an S-expression representation of programs.

Both Lisp-style atoms and universal pointers (REF ANY) were implemented in Cedar and were great successes (see discussion in footnotes 60, 61 in "A Tour Through Cedar"). However, we never did get around to defining a standard S-expression representation of programs. We had originally intended to design a representation other than the parse trees used internally by the compiler, so that the representation of parse trees could change as the compiler/language evolved without affecting tools that depended on this representation. We did not design such a representation, nor did the fallback position of using the compiler's parse trees directly prove workable. The Cedar compiler, having evolved from the Mesa compiler, did not use collectible storage, nor was it written in the safe Cedar language. It also was not organized in a way that made it easy to pull out pieces of it to use as packages.†127 Furthermore, as a result of its having evolved over several years under several implementors, the compiler had become such a monolith that changing or reorganizing it in any significant way was impractical: it would have been almost as difficult as starting over.

As a result of the difficulty of manipulating programs, tools of this type have not emerged in Cedar. However, we have begun to see a great many tools being designed and implemented which exploit the high-quality graphics interface. These tools include: an efficient, lightweight tool for checking spelling in text, a tool for creating and editing icons, a reminder service, a package for constructing a visual interface to a data structure or Cedar program interface which also allows the components to be edited or invoked.

Another use Lisp makes of the ability to treat programs as data is to provide for more general parameterization of tools and packages. For example, the conditions associated with breakpoints can be arbitrary Lisp expressions, various editor commands permit their parameters to be computed dynamically from expressions that are included in the command, etc. In principal, the Cedar interpreter makes this possible. However, this practice has not found widespread acceptance, partly because Mesa programmers simply tend to write applications in a different style than Lisp programmers, and partly because not enough attention has been given to the packaging of the interpreter in Cedar.

---

†125 Other Lisp systems such as MacLisp may have fewer facilities and organize them differently, e.g., as separate programs rather than as part of an integrated programming system, but the underlying capability that enables these facilities is still the ability to treat programs as data.

†126 Others would disagree, citing perhaps the simplicity of syntax, or the fact that all expressions are in Polish prefix notation as being equally or more important.

Inheritance/defaulting (e.g., Smalltalk subclassing)

Inheritance/defaulting was one of the items in the original catalog of programming environment capabilities [8]:

> Languages that provide for programmer-controlled defaulting or inheritance reduce the time and chance for error in the programming process by making it unnecessary to write the same code or parameter values over and over again. The basic idea is that one should be able to write programs in a way that only specifies how they differ from some previously written program. Examples include default standard values for procedure arguments (how does this call differ from a "standard" call) variant records (how does this particular record distinguish itself from the invariant part) and the Smalltalk subclass concept ... Smalltalk seems to derive considerable benefit from [subclassing]. [8]

Nothing was done about subclassing in Cedar, probably for the same reasons that most of the Lisp-related issues were not addressed: the majority of Cedar users and implementors had little or no direct experience with Smalltalk, and hence did not place as high a value upon this capability as they did on others of a more traditional Mesa flavor.†128 As a result, when a user wants a slight change or enhancement to an existing facility in Cedar, and he is unable to persuade the implementor to make the change, he simply steals the code, i.e., uses the existing program as a starting point and makes the desired changes, thereby producing his own, personalized version of the software.

The reason this works as well as it does in the Cedar environment is because one of the goals of the Mesa language is readability and maintainability by programmers other than the original implementor. The type declarations and other redundant information that may have been burdensome for the original implementor to specify when he was first constructing the program now pay great dividends. They provide a form of documentation, effectively recording certain aspects of the implementor's intent, and making the program easier to understand. Furthermore, the automatic type checking assures the borrower that when the program he has modified compiles successfully, he will not have to go through a lengthy debugging process; he has a high degree of confidence that the program will run correctly, or if there are errors, they will be localized in the area of his changes.

The disadvantage of this procedure is that when repairs or improvements are made to the original software, they do not always find their way into the modified version, unless the borrower is diligent about tracking changes. Sometimes he may be able to convince the original implementor that the modifications he has made are indeed improvements, in which case the changes may be incorporated in the original code. For example, guarded buttons (see Figure 11) were introduced into Cedar via this path. However, the danger of the proliferation of multiple, renegade versions of standard system software makes this procedure not a completely satisfactory substitute for subclassing.

---

†127 Our original strategy called for factoring the compiler into layers. For example, our plan for implementing an interpreter called for using the compiler's scanner and semantic analyzer. However, the intractibility of the compiler forced us to implement the interpreter as a separate package.

†128 Also, providing support for subclassing in Cedar would have been a very difficult task.

Polymorphism

The Cedar run-time type system allows programs to manipulate types in a fully general way. However, such programs admit the possibility of errors that are not detected at compile time but instead occur at run-time, and sometimes only under unusual circumstances. This runs counter to the Mesa philosophy that it is better to locate faults statically: "Many faults can be identified in a single run of the checker, rather than surfacing one at a time in debugging runs" and perhaps even more importantly, "Correctness is a static property of the program text; it is hard to ensure that a program that relies heavily on dynamic properties actually does what is intended" [11].

We had hoped to make types first-class citizens in Cedar; types would simply be values and could be passed as arguments and returned as results. This would enable many operations that would otherwise require run-time facilities to be expressed directly in the program text.

One area where the absence of polymorphism is most noticeable is in the treatment of lists in Cedar. A list in Cedar is a REF to a structure consisting of two fields, **first** and **rest** (the Lisp CAR and CDR). **first** contains the corresponding element of the list, and **rest** a REF to the rest of the list, i.e., its tail. If a list consists of elements of a particular type, such as INTEGER, then the type of the list is LIST OF INTEGER. If x is declared to be of type LIST OF INTEGER, the static type checking of the language guarantees that x.first is of type INTEGER, and x.rest of type LIST OF INTEGER. For example, a procedure that reverses a list of integers would be of type PROCEDURE[list: LIST OF INTEGER] RETURNS[LIST OF INTEGER]. However, the absence of polymorphism means that the programmer also has to supply a similar procedure for LIST OF REAL, LIST OF CARDINAL, LIST OF CHARACTER, etc. What is really desired is a way of defining a procedure which takes the type of the elements of a list as one of its arguments, e.g., PROCEDURE[list: LIST OF T, type: T] RETURNS[LIST OF T].

Our initial plan was to extend the Cedar language and modify the compiler, and we generated some proposals for doing this. However, as mentioned earlier, the Cedar-Mesa compiler proved to be intractable to any but very straightforward, localized modifications. The extensions necessary to include types as values definitely did not fall into this category. We therefore decided to postpone any further incremental changes to the compiler, and instead to design a generalized version of the Cedar language called PolyCedar which would incorporate a number of the ideas found in languages like Russell [4]. However, such a project requires a substantial effort that we have not yet been able to mount.

Document Editing, Editor Integrated with Language System

The original EPE report stated:

> Editing is just one function of a language system, carried out using a particular sublanguage. As such, it should be integrated with the rest of the language system in that: (1) the user doing editing can call on arbitrary programs to compute commands or data needed for the editing process, including the ability to pass selections from the thing being edited to the computation as arguments; (2) any program can call on the editor as a package. The latter seems very useful and relatively easy to achieve. We agree the former is also valuable, but there is disagreement over whether it is merely valuable or extremely important. [8]

Cedar users in general agree that Tioga has been an outstanding success both as a text and program editor, when viewed as a package *invoked by the user.* However, Tioga has not achieved the degree of integration we aspired to in the original EPE report. Many operations that the user can perform on a document cannot be conveniently performed by a program. Although there is a program interface to the Tioga editor, it requires for most of its operations that the implicit argument be the current selection. Thus, programs cannot operate on documents directly and invisibly, but must effectively simulate the actions of a user moving the selection around, and obtain the results of these operations by examining the current selection after the operations complete. As a result, there is considerable extraneous (from

the user's standpoint) screen activity while such an operation is being performed, e.g., selections changing, viewers scrolling, etc., and furthermore, the user cannot be performing at the same time any operations that affect the display, such as clicking the mouse or typing characters. Both of these factors tend to limit the utility of using Tioga as a callable package to perform editing operations.

What Next?

There is no good objective way to evaluate a programming environment, no way of certifying that it "works." There are no solutions only choices. The previous section discussed some of the things we decided not to pursue in Cedar, and the shortcomings that resulted. This section briefly lists some tasks that, in the author's opinion, might prove fruitful to attack next.

Access to on-line documentation (Helpsys)

> Good on-line documentation, both for reference and for learning, can greatly reduce the need for time spent studying an enormous manual, can provide instant cross-linking of related subjects in a way that hardcopy cannot, and can use one's current context to implicitly locate relevant material. Interlisp's Helpsys facility is unique in these respects. However, creating and maintaining such documentation is a tremendous amount of work, even if the process is partly automated. [8]

No work has been done yet in Cedar on providing convenient access to on-line documentation of the type available in Interlisp or Smalltalk. In fact, Cedar suffers from a lack of adequate documentation in general. Even where material is documented, it is often hard to find due to lack of coherent organization: users have to know where to look. Providing good documentation for Cedar will be one of the highest priority items on our agenda of things to do next.

Masterscope

Masterscope is an interactive program for analyzing and cross-referencing user programs in Interlisp. We recognized the importance and utility of such a facility for Cedar, and envisioned that it would be one of the principal clients of the Cedar data-base facility. However, due to various priorities and limited resources, nothing has been done about Masterscope yet, a shortfall that Cedar users often bemoan. For example, when the author polled the Cedar community for examples of definitions of new Viewer Classes, one user responded: "Here's one where I could really use a 'global' Masterscope... I've implemented so many viewer classes I'll probably forget some, and it would be very tedious to examine all my code by hand."

Implementing a Masterscope-like facility for Cedar would be considerably simplified if there were a standard, program-manipulable representation of Cedar programs (see earlier discussion under "Easy Use Of Programs as Data"). As is, before one could begin worrying about analyzing a Cedar program, it would be necessary to implement a facility for parsing the text of a Cedar program into a structure which was program manipulable.

Altering the Flow of Control from the Debugger

The ability to alter the flow of control from within the debugger would partially offset the lack of support for fast turnaround for small program changes by allowing the user to simulate the effects of a change by manually altering data and control from the debugger. In this way, the user would be able to see whether the rest of his program would operate in the desired fashion if he made a particular change, without having to make the change and start over.

The simplest form of such a facility is the ability to stop a program at the entry to a procedure, execute the procedure, examine its return value(s), and specify different ones. We can do this now in

Cedar, but only for certain cases. Even more useful would be the ability to stop at a breakpoint on the entry to a procedure, execute the procedure, examine its values/effects, change its arguments, and try again. Interlisp provides a more general capability which allows the user to unwind the stack back to an arbitrary location from within the debugger. This is particularly useful when a problem is detected after the fact. Such a facility would be similar to Mesa's existing signal-handling mechanism, but requires being able to generate signals and construct catch phrases at run-time. We believe that such an extension to Cedar is straightforward, though non-trivial.

Recompiling Interfaces

One of the great strengths of the Mesa programming language is the explicit notion of an interface. Separation of Mesa programs into interfaces and implementations of these interfaces enable implementors and clients to work independently, and to make changes independently, as long as they respect the interface. However, the present need for recompilations of a large number of files whenever a fundamental interface is changed *in any way* is a weakness in Cedar. Not only does every program that depends on the interface need to be recompiled, but if any other interfaces depend on the interface, they, and all of their clients, must also be recompiled before the system is once again in a consistent state. The existence of this ripple effect makes changing a basic interface a major undertaking requiring precise coordination. As a result, there is considerable social pressure to freeze program interfaces in Cedar, often before we have had sufficient opportunity to experiment with the interfaces.

One improvement that would significantly improve the situation would be to eliminate the need for recompilation when an interface is changed in an upwards compatible fashion. The two most common examples of such a change are changes to the comments in an interface and the addition of new items to an interface.

Because of the ready accessibility of all program sources in Cedar via the version map, interfaces are often their own most frequently used documentation. "A Tour Through Cedar" illustrated how the user could readily see not only the type declaration for any item in an interface, but also the comments associated with the item. Since documentation often needs to be debugged as much as programs, we would like to be able to modify or extend the comments in a fundamental interface without introducing the large ripple effect that follows such a recompilation. Currently, Cedar uses as the version stamp for an interface the date and time the interface was compiled. One proposal for allowing comments to be changed is to *compute* the version stamp, based on the contents of the interface minus the comments.

A much more ambitious change would be to keep version stamps in an interface on an item by item basis. In this case, programs would have to be recompiled only when the particular items that they actually used from a given interface were changed. This would also allow, as a special case, the addition of new items to an interface without affecting existing programs.

Inter-language communication

The section "Support for Wide Range of Binding Times" discussed the desirability of being able initially to implement a program using late binding to defer various binding decisions in order to minimize the constraints on the program, and thereby decrease its resistance to change, and then later to bind these decisions earlier in the program to provide for increased efficiency and reliability. Perhaps the first such decision that a programmer has to make is what programming language to use. The ultimate in delayed binding would be to enable the programmer to change this decision for those parts of his program that needed it, for example, initially writing an application in Lisp, and then optimizing those parts that need it by recasting them in Mesa. This is an extremely important and challenging area for future research.

Conclusion

> Today, in the fall of 1983, Cedar is a reality and, by common consent, one of the most advanced programming environments in the world. Visitors from other laboratories are envious of Cedar's emphasis on visual interaction; of its support for concurrent tasks; of its sophisticated debugging facilities. Above all, the environment does indeed make it possible for a researcher to design and implement an experimental computer system and get it used and tested in a remarkably short period of time. For instance, various programmers using the Cedar environment have in the last few months been able to build and test two experimental systems for handling electronic mail, one for computer storage of voice messages, one for producing raster images, several for VLSI design. They all report favorably on the ease with which they could knock together real, functioning systems.
>
> The Cedar project is, however, still far from complete. ... There are still many problems to be solved, and we have not yet succeeded in meeting all of our goals. We need even more memory, both real and virtual. We need a more flexible type system. We do not yet have integrated access to databases. We want better ways of manipulating information from source programs. We need much faster turnaround for program changes than we have yet been able to achieve. And as Cedar begins dramatically to increase our ability to build complicated systems, we have come to feel the need for better tools to describe and control them. [6]

The Cedar project has been an unusual one from several standpoints. To the author's knowledge, in no other case have the goals of an environment been so clearly stated, even before the first line of code been written, and in very few cases have these goals been so ambitious. The successes of Cedar, how much effort was actually required to achieve them, the shortfalls of Cedar, why they occurred and the effect they had on the resulting environment, all have much to teach us about the design and implementation of large programming environments, an area of endeavor that is becoming increasingly more important as the plummetting price of computer hardware makes powerful personal workstations increasingly accessible to programmers.

Appendix 1: Catalogue of Programming Environment Capabilities

The following list of desirable capabilities for a programming environment was compiled by the first Experimental Programming Environment Working Group. The complete results of their findings are contained in [7].

Virtual machine/programming language

Large virtual address space (> 24 bits)

Direct addressing for files

Well-integrated access to large, robust data bases

Memory management–object/page swapping

Object management–garbage collection, reference counting.

Some support for interrupts

Adequate exceptional condition handling

User access to the machine's capability for packed data

Program-manipulable representation of programs

Run-time availability of all information derivable from source program (e.g., names, types, scopes)

Statically checked type system

Self-typing data (*a la* Lisp and Smalltalk), run-time type system

Encapsulation/protection mechanisms (scopes, classes, import/export rules)

Abstraction mechanisms; explicit notion of "interface"

Non-hierarchical control (coroutines, backtracking)

Adequate run-time efficiency

Inter-language communication

Uniform screen management

Inheritance/defaulting

Ability to extend language (e.g., operator overloading)

Ability to create fully integrated local sublanguages

User access to the machine's capability for multi-precision arithmetic

Good facilities for processes, monitors, interrupts

Simple unambiguous syntax

Control over importation of names

User packages as "first-class" citizens

Closures

Full-scale inter-language communication

User microprogramming
Clean data and control-trapping mechanisms
"Good" exceptional condition handling

Tools

Fast turnaround for minor program changes (less than 5 seconds)
Compiler/interpreter available with low overhead at run time
Cross-reference/annotation capability
Prettyprinter
Consistent compilation
Version Control
Librarian, program-oriented filing system (including Browser)
Source-language debugger
Dynamic measurement facilities
Checkpoint, establishing a protected environment
History and undoing
Editor integrated with language system
More optimizing compiler if user willing to bind more tightly–with full compatibility
Aids for incremental development (stubs, outstanding task list)
Regression testing system
Random testing aids
(high capability) Masterscope
Access to on-line documentation (Helpsys)
Static analyzers; verifier, performance predictor

Packages

Text objects and images
Line objects and images
Scanned (bitmap) objects and images
Formatted document files
More elaborate screen management
Remote file storage
Small data base manager
Message transmission system
Remote procedure call
Event logging

Background processing

Generalized cache

Document editing

Forms

Menus and other standard user interfaces

History lists

User access to full bandwidth of disk

(English) dictionary service

Teleconferencing

Audio

User access to full bandwidth of networks

Other

Adequate reference documentation

"Efficient" interface for experts

Uniformity in command interface

"Self-teaching" interface for beginners

Good introductory documentation

Appendix 2: Prioritized Catalogue of Programming Environment Capabilities

The following is the same list that appears in Appendix 1, sorted by the priorities assigned to each capability by the EPE Working Group.

Priority A

Object management–garbage collection, reference counting.

Statically checked type system

Memory management–object/page swapping

Abstraction mechanisms; explicit notion of "interface"

Fast turnaround for minor program changes (less than 5 seconds)

Adequate run-time efficiency

Large virtual address space ($\geq$ 24 bits)

Priority B

Encapsulation/protection mechanisms (scopes, classes, import/export rules)

Well-integrated access to large, robust data bases

Self-typing data (*a la* Lisp and Smalltalk), run-time type system

Consistent compilation

Version Control

Source-language debugger

Text objects and images

Uniform screen management

User access to the machine's capability for packed data

Run-time availability of all information derivable from source program (e.g., names, types, scopes)

Priority C

Direct addressing for files

Some support for interrupts

Compiler/interpreter available with low overhead at run time

Adequate reference documentation

Librarian, program-oriented filing system (including Browser)

Program-manipulable representation of programs

Dynamic measurement facilities

Scanned (bitmap) objects and images

Formatted document files

"Efficient" interface for experts

Line objects and images

Remote file storage

Priority D

Inter-language communication

History and undoing

Non-hierarchical control (coroutines, backtracking)

Ability to extend language (e.g., operator overloading)

Ability to create fully integrated local sublanguages

Closures

Checkpoint, establishing a protected environment

Inheritance/defaulting

Cross-reference/annotation capability

Prettyprinter

Menus and other standard user interfaces

Document editing

Adequate exceptional condition handling

Editor integrated with language system

Remote procedure call

More optimizing compiler if user willing to bind more tightly–with full compatibility

Access to on-line documentation (Helpsys)

Message transmission system

Event logging

Generalized cache

Forms

Uniformity in command interface

References

[1] Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder. "Grapevine: An Exercise in Distributed Computing," *Communications of the ACM*, Volume 25, Number 4, April 1982, pp 260-274.

[2] Andrew D. Birrell, and Bruce Jay Nelson. "Implementing Remote Procedure Calls," CSL-83-7, October, 1983 (also in *Transactions on Computer Systems*, Volume 2, Number 1, February, 1984).

[3] D.G. Bobrow, and M. Stefik, "The Loops Manual," Knowledge Systems Area, Xerox PARC 1983.

[4] H. Boehm, A. Demers and J. Donahue, "An Informal Description of Russell," Technical Report TR80-430, Computer Science Department, Cornell University, 1980.

[5] R. G. G. Cattell, "Design and Implementation of a Relationship-Entity-Datum Data Model," Xerox Palo Alto Research Center Report CSL-83-4. May, 1983.

[6] Cedar Implementors, "The CSL Cedar Project," *Update*, Xerox Palo Alto Research Center Technical Information Center, January 4, 1984.

[7] Douglas W. Clark, Butler W. Lampson, and Kenneth A. Pier, "The Memory System of a High-Performance Personal Computer," Xerox Palo Alto Research Center Report CSL-81-1. January, 1981 (also in *IEEE Transactions on Computers*, Vol. C-30, No. 10, pp. 715-733, October, 1981).

[8] L. Peter Deutsch and Edward A. Taft, "Requirements for an Experimental Programming Environment," Xerox Palo Alto Research Center Report CSL-80-10. June, 1980.

[9] L. Peter Deutsch and Daniel G. Bobrow, "An Efficient, Incremental, Automatic Garbage Collector," *Communications of the ACM*, Volume 19, Number 7, July, 1976.

[10] Adele Goldberg and Dave Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[11] J. J. Horning, editor, "Report from Second Programming Environment Working Group," Internal Memo, December 13, 1978.

[12] J. J. Horning, "Cedar Language Overview," Internal Memo, 1983.

[13] D. H. Ingalls, "The Smalltalk-76 Programming System: Design and Implementation," *Proceedings of the 5th Annual ACM Symposium on Principles of Programing Languages*, January 1978, pp 9-16.

[14] *Interlisp Reference Manual*, Xerox Corporation, revised October, 1983.

[15] Butler W. Lampson and Kenneth A. Pier, "A Processor for a High-Performance Personal Computer," Xerox Palo Alto Research Center Report CSL-81-1. January, 1981 (also in *Proceedings of Seventh Symposium on Computer Architecture*, SigArch/IEEE, La Baule, May 1980, pp. 146-160).

[16] Butler W. Lampson, "Hints for Computer System Design," *ACM Symposium on Operating System Principles*, October, 1983.

[17] Larry Masinter, "Global Program Analysis in an Interactive Environment," Xerox Palo Alto Research Center Report SSL-80-1, January, 1980.

[18] John Maxwell, "The Cedar Spy," Internal Memo, 1983.

[19] Scott McGregor "The Viewers Window Package," Internal Memo, 1983.

[20] James G. Mitchell, William Maybury, and Richard Sweet. "Mesa Language Manual." Version 5.0. Xerox Palo Alto Research Center Report CSL-79-3, April 1979.

[21] James Morris, "The Cedar Project," Internal Memo, March 31, 1980.

[22] Bill Paxton, "The Tioga Editor," Internal Memo, 1983.

[23] Lyle Ramshaw, "The Briefing Blurb," Internal Memo, 1983.

[24] David Redell, et. al., "Pilot: An Operating System for a Personal Computer", *Communications of the Association for Computing Machinery*, Vol. 23, No. 2, February, 1980.

[25] Erik Sandewall, "Programming in an Interactive Environment: the 'Lisp' Experience," Computing Surveys, Vol. 10, No. 1, March 1978.

[26] Eric Emerson Schmidt, "Controlling Large Software Development In a Districtubed Environment," Xerox Palo Alto Research Center Report CSL-82-7, December, 1982.

[27] Beau Sheil, "Environments for Exploratory Programming," *Datamation*, Vol. 29. No. 2, February, 1983, pp 131-144.

[28] M. Stefik, D.G. Bobrow, S. Mittal, and L. Conway, "Knowledge Programming in Loops: Report on an Experimental Course," *AAAI Magazine*, Vol. 4, No. 3, Fall 1983.

[29] Warren Teitelman, "A Display Oriented Programmers Assistant," *Proceedings of the Fifth International Joint Conference on Aritifical Intelligence*, August, 1977 .

[30] Warren Teitelman, *The Interlisp Reference Manual*, revised 1978, Xerox Palo Alto Research Center.

[31] Warren Teitelman and Larry Masinter, "The Interlisp Programming Experience," *Computer*, Vol. 14, No. 4, April 1981, pp 25-33.

[32] C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, D. R. Boggs, "Alto: A Personal Computer," CSL-79-11. August, 1979 (also in *Computer Structures: Readings and Examples, second edition*, by Siewiorek, Bell and Newell).

[33] John Warnock and Douglas K. Wyatt, "A Device Independent Graphics Imaging Model for Use with Raster Devices," *Computer Graphics*, Volume 16, Number 3, July, 1982.