A Tour Through Cedar

Warren Teitelman

Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304

Introduction

This paper†1 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 [8] 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 the 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. Comments that would be distracting during a live demonstration but are appropriate for the paper are included as footnotes.†2

Background

In 1977, the computing community at Xerox Palo Alto Research Center (PARC) consisted of three distinct cultures: Interlisp [15], Smalltalk [6], and Mesa [11]. Both the Smalltalk and Mesa communities programmed primarily on the Alto, a small personal computer that had been developed at PARC [16]. The Interlisp programmers operated on a time-shared, main-frame computer. 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.

We decided to solve these problems by designing and building a much more powerful personal computer, the Dorado [7]. The arrival of the Dorado in 1978 resolved our immediate hardware problems of execution speed, memory size, and address space. Our ability to experiment with computer systems, critical to our research, was now limited only by our programming capabilities of which the principal component was the programming environment.

Thus, in 1978 we embarked on a project to design and implement Cedar, an advanced programming environment that would take advantage of the capabilities provided by the Dorado. A group of us met for several months and produced a catalogue of programming environment capabilities for such an environment [4]. At this point, we were not committed to either of the three principal programming languages in use at PARC, but after lengthy deliberations we decided to base Cedar on Mesa [14]. However, our goal was that Cedar would support each of the Lisp, Mesa, and Smalltalk programming styles.

For various reasons as the Cedar project evolved, some of our goals changed, or at least were reprioritized; a fair characterization of the Cedar project as it is currently constituted is that it is an attempt to take the Mesa language and build for it a programming environment based on ideas and techniques from Interlisp and Smalltalk.

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.†3 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

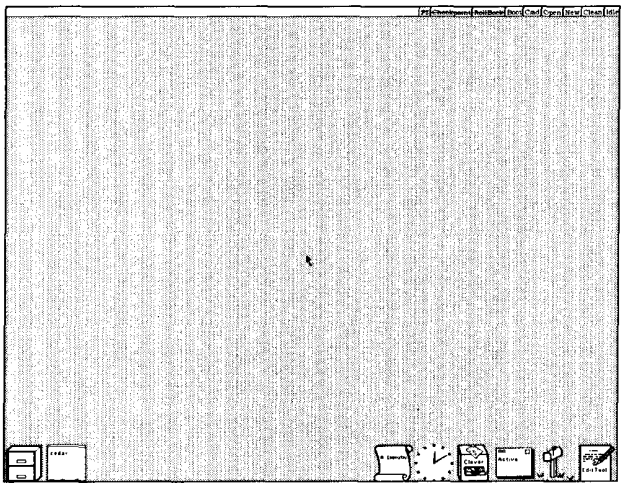

Figure 1: Initial cedar screen layout showing various icons

†1 This paper is a condensed version of a paper contained in [14]; because of space limitations, many of the figures have been removed and comments have been abbreviated or eliminated. [14] also contains a second paper entitled "The Roots of Cedar," which describes the conditions in 1978 that led us to embark on the Cedar project, and helped us to define its objectives and goals, as well as a third paper entitled "Cedar: The Report Card," which evaluates the successes and failures of Cedar.

†2 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.

†3 All Dorados have 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).

0270-5257/84/0000/0181$01.00©1984 IEEE

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 in Figure 1 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 down the hall whose name is Clover, and the icon in the left hand corner of the display that looks like a file cabinet views the FileTool, a facility for obtaining files from remote servers.†4

Viewers Window Package

The Viewers Window Package provides the basic display paradigm for Cedar [10]. It allows users and programs to create, destroy, move, and resize individual rectangular 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 facility are called *tools*, and viewers that simply invoke a procedure are called *buttons*.†5 The FileTool and the Walnut Mail Reader shown at the bottom of Figure 1 are examples of tools, and the nine small boxes labelled 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 – 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 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 [16]. 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 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.

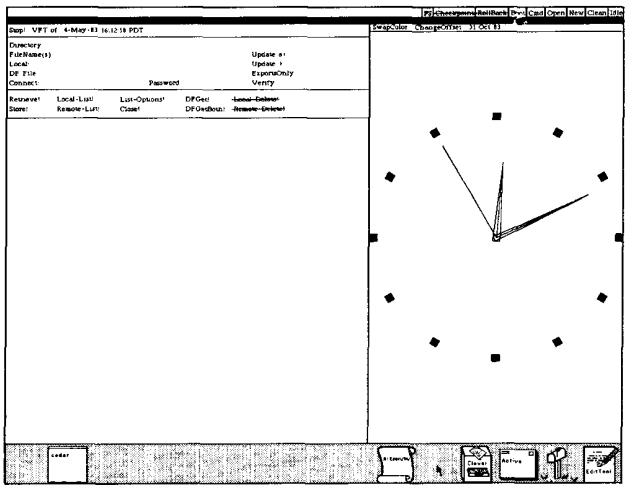

Figure 2: The same display after opening the FileTool and Clock viewers

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, the Clock has a menu that includes the buttons SwapColor and ChangeOffset (see Figure 2). More elaborate menus are associated with text viewers, as shown in Figure 6.

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 Figure 10). 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 hard copy 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 now occupied by the FileTool and Clock viewers is divided into two columns.†6 When more than one viewer is created or opened in the same column, the viewers automatically share the space available.†7 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

†4 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).

†5 The principal difference between a tool and a button 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 invokations), whereas a button may take an argument, but essentially performs the same operation each time.

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

†7 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. On the other hand, 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 to access 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.

that column are automatically closed. To show you how this works, I'll open the remaining icon on the left side of Figure 2, the one labelled "Cedar" that looks like a chalkboard with erasers on its ledge. This produces the arrangement shown in Figure 3.

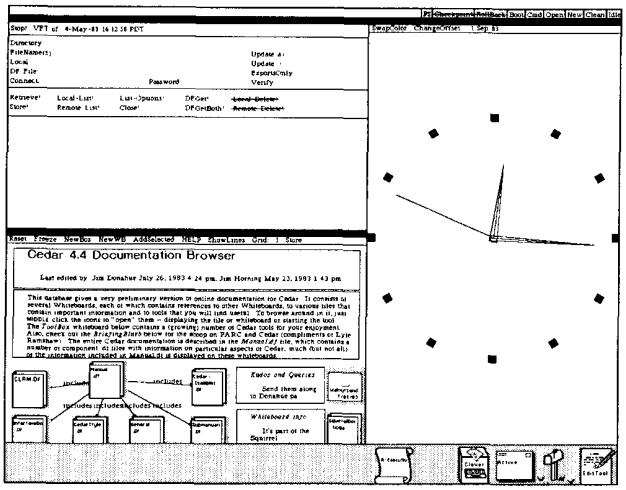

Figure 3: The FileTool and Documentation Browser share the left column

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, i.e., whiteboard viewers provide a spatial way of organizing data. 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 4.

In order to see more of this whiteboard viewer, let's move the FileTool from the left-hand column to the right-hand column using the appropriate button in the menu that is hidden under the FileTool's caption. This produces the screen layout shown in Figure 4.

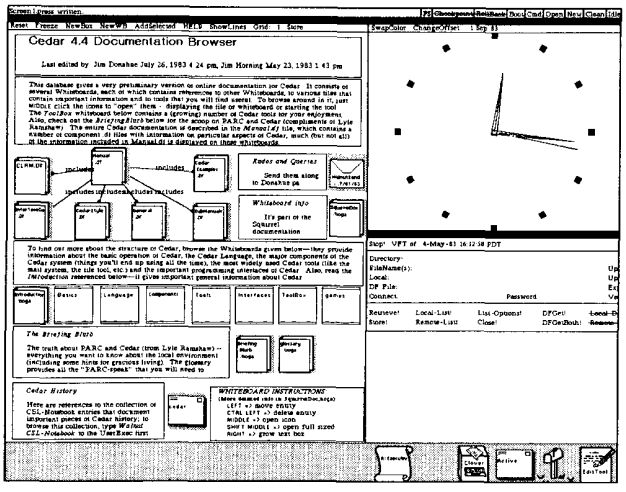

Figure 4: The Cedar Documentation Browser

Online Documentation

The Cedar Documentation Browser shown in Figure 4 uses a whiteboard viewer to display a data base for the online documentation for Cedar [3]. 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 hand 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.†8 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 5.

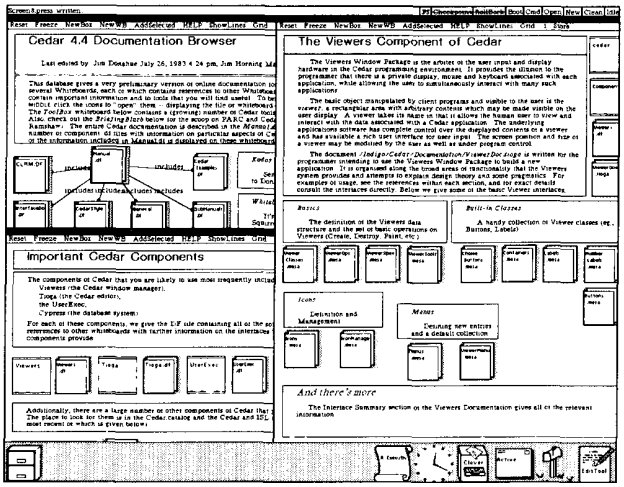

Figure 5: Browsing the documentation using whiteboards

The whiteboard for the Viewers Package that appears on the right in Figure 5 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.†9

The Tioga Editor and Document Preparation System

The text viewer that appears in the left-hand column of the display in Figure 6 is the on-line documentation for the viewer package itself, in the form of a Tioga document. Tioga is both the editor for Cedar programs as well as its document preparation system.†10 In Tioga, a document is a tree structure of nodes rather than a list of paragraphs

†8 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).

†9 We have placed a great deal of emphasis in the design of Cedar on uniformity of command interface. "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" [4].

†10 The Viewers Package documentation shown in Figure 6, as well as all Cedar documentation, was prepared using the Tioga editor, as was the paper that you are now reading. When hardopy is needed, the Tioga typesetter (represented by the printer icon shown at the bottom of Figure 6) is used to generate high quality hardcopy from the document and send it to the corresponding printer.

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 6 only the top level of nodes are shown, thus effectively providing a table of contents.

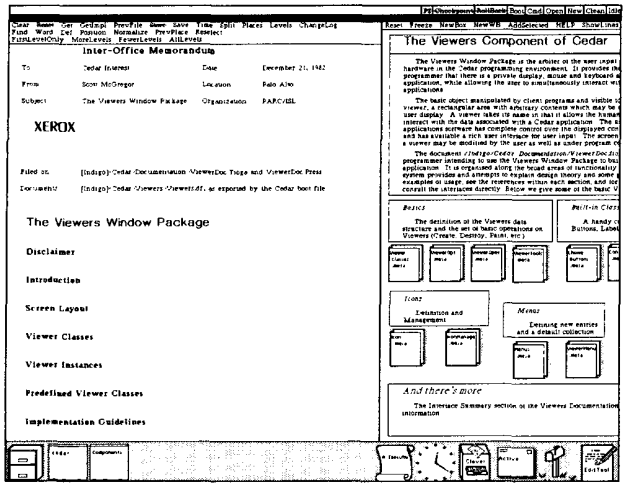

Figure 6: Online documentation for the Viewers Window Package

In combination with scrolling, the use of levels in Tioga makes it easy for the user to browse through a document or program and quickly find the part that interests him. For example, let's scroll to the section entitled "PreDefinedViewer Classes"†11 and click the MoreLevels menu button at the top of the viewer.†12 This allows us to see one more level of detail, the titles of subsections, as shown in Figure 7.

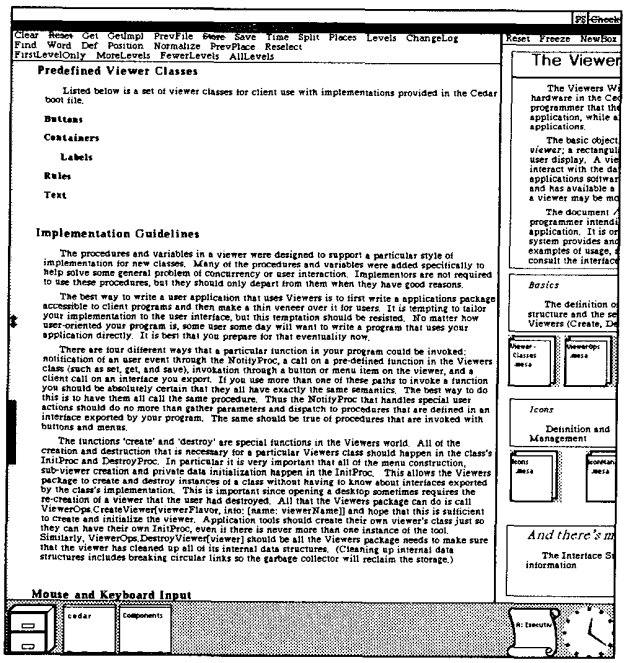

Figure 7: Browsing a Tioga document using level clipping to suppress detail

†11 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 scrollbar is visible in Figure 7.

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 node plus all of its children). For example, I can select the word "environment" in the first paragraph of the introduction (see Figure 8) 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." Tioga indicates the current insertion point on the display by the appearance of a blinking caret. (In Figure 8, 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 8 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."

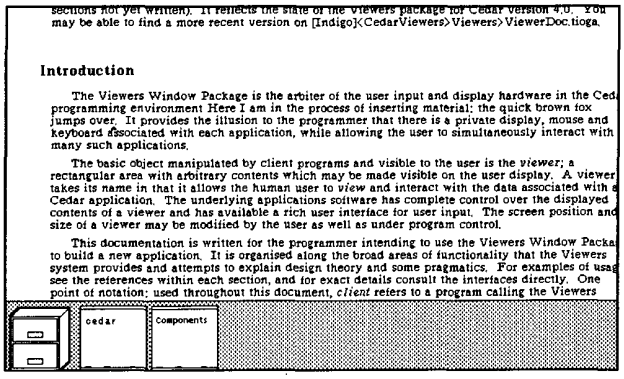

Figure 8: Inserting characters into a Tioga document

Commands can be given to Tioga using various control keys, e.g., typing a character while the CTRL key is depressed. For example, I'll *undo* the insertion I just made with a single keystroke. This ability to undo 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*. 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 (see Figure 9).

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, e.g., editing, compiling, reading mail.†13

†12 The 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.

†13 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" [4]. This is especially important in an environment such as ours where the use of personal machines encourages using the time when the user is thinking or the time between keystrokes by performing various background tasks, 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.

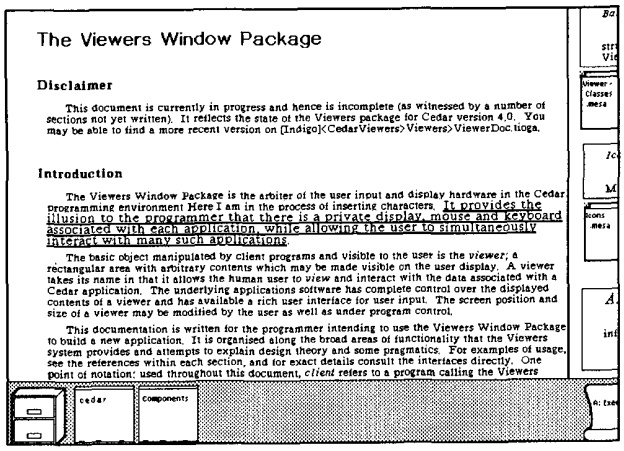

Figure 9: Changing fonts

We aren't going to be needing this viewer, so let's destroy it using the Destroy menu button which is contained in menu that is hidden under the caption. We do this by moving the mouse into the caption area of the viewer, which causes the caption menu to be displayed as shown in Figure 10. Notice that the Destroy button has a line through it. This indicates that the button is *guarded.* Guarded buttons must be clicked twice in a short time interval to take effect.†14 This 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. If we were to go ahead and destroy this viewer anyway, a new icon labelled UnsavedDocuments List would be created. If I were to open this icon, I would see: "The following files were edited but not saved. They may still be restored with edits intact simply by loading them." i.e., I could still get my edits back if I really wanted them.

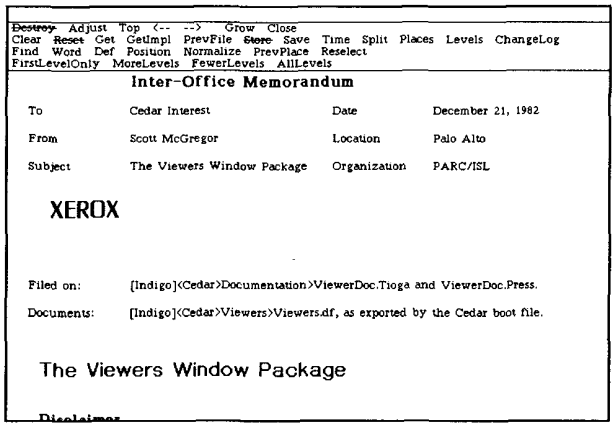

Figure 10: The Destroy menu button is guarded to prevent accidents

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.

†14 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.

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. Commands are typed to the executive by typing to its viewer, and its output 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 6 that looks like a scroll and is labelled "A: Executive." I'll open this icon in the usual way, and then move the mouse into the resulting viewer and click it. The executive is now listening to me, i.e., it will see the characters that I type.

The 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.
```

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.†15

```
&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 produced a file which was reasonably close in spelling. This 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 [13].

The UserExec has loaded and started the Watch program, which created an icon for the Watch tool. I'll open the Watch icon, and we'll observe the Watch tool in action as I execute another event in the UserExec (see Figure 11).

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

†15 Since Cedar programs are often written in expectation of production use of the program, performance monitoring and tuning is an important issue. The Watch tool is just one example of a number of such tools available in Cedar. For example, 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" [9].

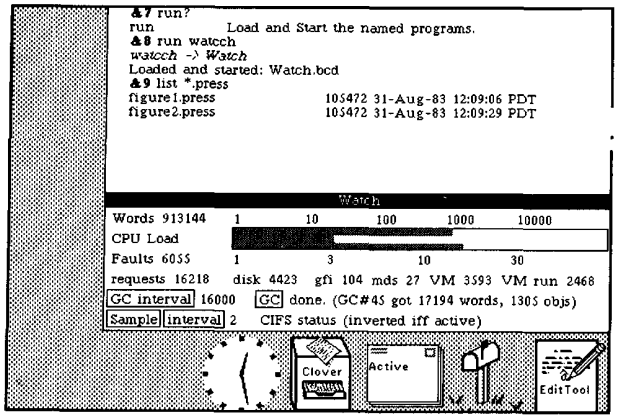

Figure 11: The Watch Tool monitors program activity

totally compiled.†16 Thus, the Cedar interpreter is an important and integral part of the Cedar environment, despite the fact that Cedar is a compiler-oriented language.

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. (This menu can be seen in Figure 15.)

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
&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. For example, let's multiply it by 1.4:

```
&3 ← &1 * 1.4
9.8
```

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.†17

```
&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.†18

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 ← &?                                    †19
is of type System.GreenwichMeanTime
```

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.

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. 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.†20 Atoms, which are very similar to Lisp atoms, and Lists are also examples of REFs.†21

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)
&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.

Manipulating Lists

The List interface includes a variety of procedures for manipulating lists, such as Append, Reverse, Remove, Union, and Intersection.

†16 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.

†17 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.

†18 The notion of abstraction mechanisms and the explicit notion of interface was an important item in our original catalogue of programming environment capabilities [4]: "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" [4]. 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 module level as is currently done).

†19 The value of the variable & is the value of the last event execute, i.e., in this case & and &7 have the same value. †20, †21 (see next page)

Let's try the procedure Reverse on the list constructed in event 9.

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

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.†22 DWIM found a procedure, Reverse, whose spelling was pretty close to what I typed, and in Figure 12, is now waiting for me to confirm or reject the correction, 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.†23 When (and if) I confirm the correction, the corrected expression will be evaluated.

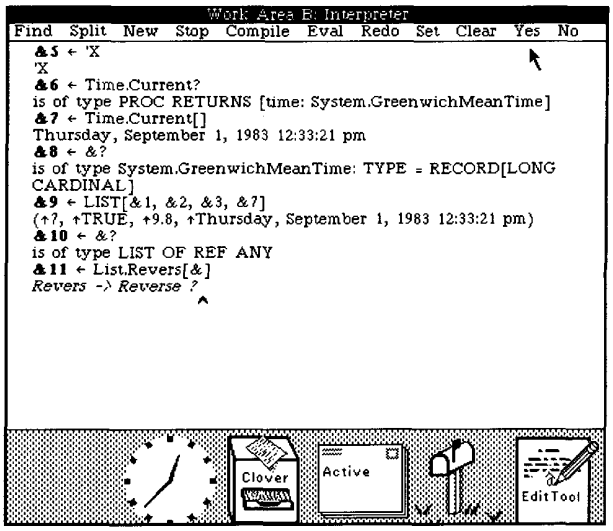

Figure 12: Confirming a DWIM error correction

&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.†24 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

---

†20 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. "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" [12].

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.

†21 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 polymoprhism is cited as the biggest shortcoming of the current implementation.

†22 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 in 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: 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.

†23 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 handle gracefully the issue of typeahead 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 confirmation. The desired behavior of the system is that the user be able to confirm without having his typeahead 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 typeahead.

†24 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 alter the characters it contains.

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 13). Such a selection is called a *source* selection.

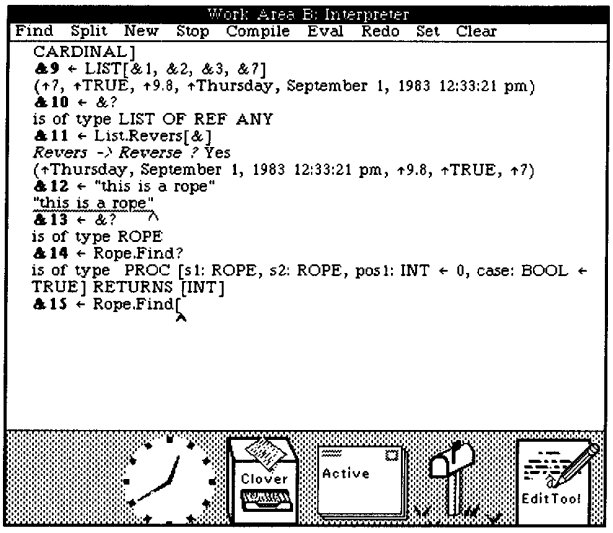

Figure 13: Copying displayed characters instead of typing them

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.†25

```
&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 create a viewer on this interface and see if there are any comments associated with this procedure. I select Rope.Find in my Work Area, and then click the Open menu button in the message area at the top of the screen. This tells the Viewers Package to create a new viewer, 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 14.

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

†25 This feature is tremendously useful. It greatly increases the bandwidth of the user's interaction with the system. It also enables us to use 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 an asset: the name contains so much information it is in effect a form of documentation.)

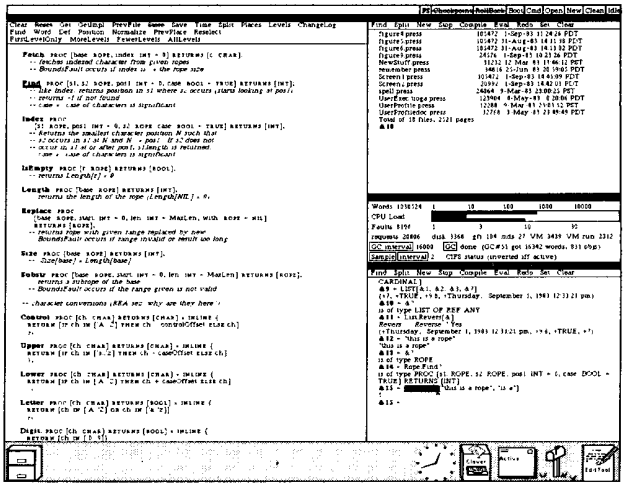

Figure 14: Access to program sources provides online documentation

file, scroll to the appropriate location, and then select the place where I want the breakpoint inserted.†26 I then plant the breakpoint by clicking the Set menu button in my Work Area.†27

```
&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.
```

The breakpoint has been set.†28 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 15, 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 in the Work Area and clicking the Redo menu button. The UserExecutive maintains a history of the events that have been executed.†29 It uses this history list to find the event corresponding to my selection and reexecute it.†30

†26 The reader may wonder how I knew where to place the breakpoint. In this particular case, I happen 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, and examining data. This behavior is facilitated by the structuring of the source files that Tioga enables, and by the use of long, suggestive names. 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.

†27 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.

†28 Setting a break point 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.

†29 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, 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.

†30 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.*

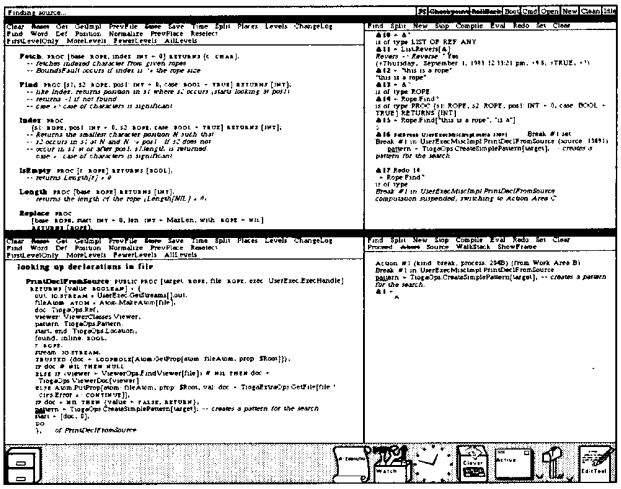

Figure 15: Hitting a breakpoint

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.†31 In Figure 15 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 click the ShowFrame menu button in my Action Area.

&1 ShowFrame args UserExecMiscImpl.PrintDeclFromSource
A- target: "Find\n", †32
file: "Rope.mesa",
exec: {UserExecHandle: "B"} †33

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? I'll reset target using the interpreter.

&2 ← target ← "Find"
"Find"

†31 This method also supports the Principle of Non-Preemption espoused in footnote 13. 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.

†32 \n is how Cedar prints carriage-return when it appears as data.

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

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.

&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.

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 the call stack one frame.

&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 viewer on the left.†34

&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*

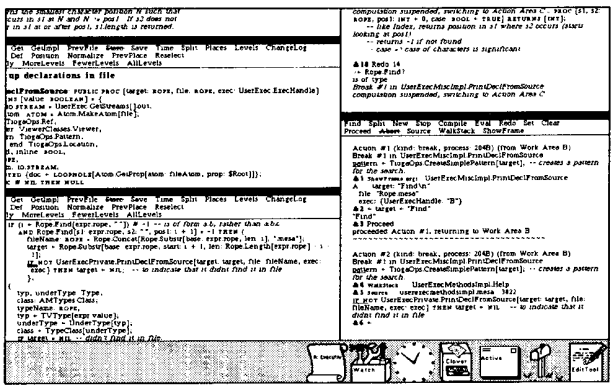

Figure 16: The Source command finds the location in the source file

The underlined location in the viewer on the left in Figure 16 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 that immediately before this statement is the expression: target ← Rope.Substr[base: expr.rope, start: i + 1, len: Rope.Length[expr.rope] – i – 1]. This expression

†34 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 4), the file will be automatically obtained from a file server.
~~~~~~

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, and the value of len, the length of the substring.

```
&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 (see Figure 17) 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. I fill in the comments field, and then save the file using the Save menu button.

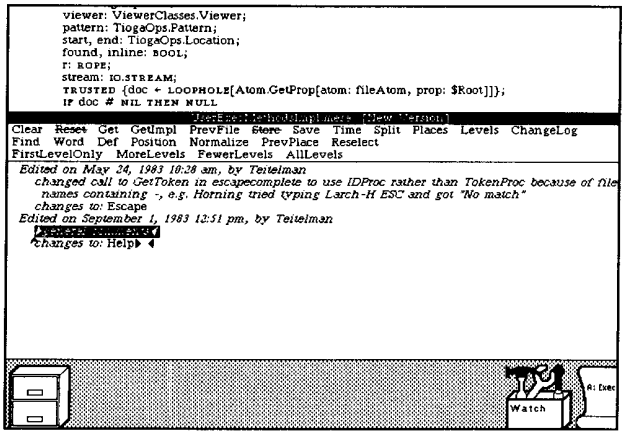

Figure 17: Automatic Change Log maintenance

Now let's go back to the Action Area on the right, 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.

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.†35

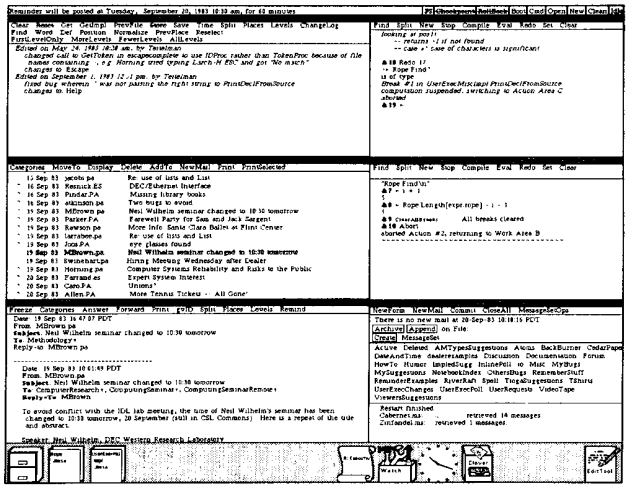

Figure 18: The Walnut Mail System; Entering a message into my reminder system

The messages marked with ? (see Figure 18) are the ones that I haven't read yet. 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 shown in Figure 18. The Reminder system obtains the time and date for the corresponding event from the message itself,†36 and when the corresponding time rolls around, a blinking icon will be automatically displayed on my screen (as shown in Figure 19). If I open this reminder icon, the viewer will contain this message.†37

Messages that I want to save so that I can refer to them later I frequently sort into various categories which I have created called *message sets*. 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 18): BackBurner, CedarPaper, Discussion, Documentation, etc.

---

†35 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 18 (middle 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 [3] to maintain information about stored messages.

†36 You can see the feedback from the reminder system 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).

†37 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 highest priority items in our original Catalogue of Programming Environment Capabilities [4].

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.

The message that I am interested in is 11-Feb-83 Willie-Sue.pa bug??.[†38] 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 19 (top viewer, left hand column).

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. Instead of typing the comment in the message into Work Area A, I can simply copy it by selecting the corresponding characters in the message while holding down the SHIFT key.[†39]

&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

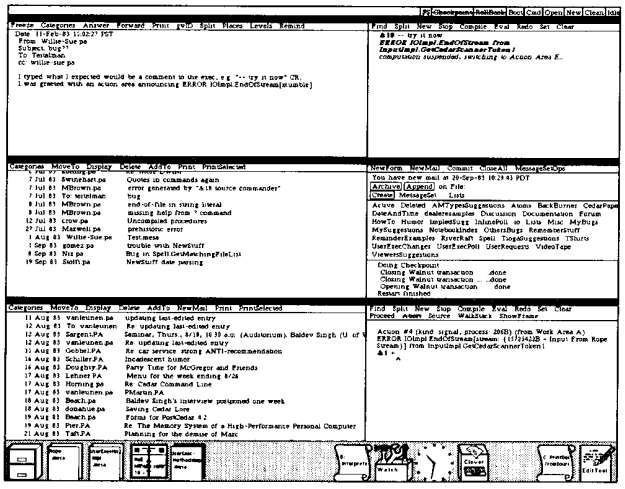

Figure 19: A user reports a bug via an electronic message; Checking out the problem causes an error which results in the creation of a new Action Area

Well, it's just like the message said. We got an EndOfStream error, and are now in a new Action Area.[†40] [†41] Let's walk the stack and see what's happening.

&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;[†42]

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.

&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.[†43] 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.

&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 20.

---

†38 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.

†39 Note that in this case I am 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!

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

†41 Notice that, since it is now 10:30AM, the reminder concerning that talk I wanted to attend (entered in Figure 18) has popped up at the bottom of my display, fourth icon from the left. The icon is blinking to call itself to my attention. If I were to open this icon, I would find the original message.

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

†43 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 typical client program can manipulate a stream without regard to the class that implements it, so varying stream implementations can be substituted without effect on the program [2]. Furthermore, new implementations of streams can be supplied by the user. 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.

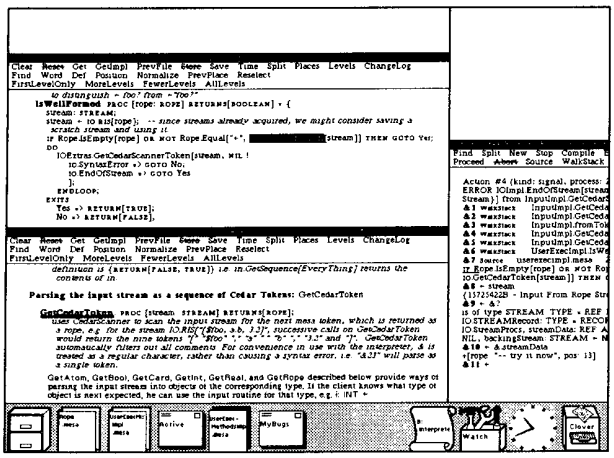

Figure 20: The IO interface serves as online documentation

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.†44 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.

Compiling, Support for Parallel 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 21. I can instruct the system to compile all of the files that need compilation via the command compileall.†45

```
&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 21, I am in the process of composing my answer in the viewer on the lower left, while the compiler continues to run in the Work Area at the upper right.†46

I finish composing the message, and click Send, and the message is sent on its way. In Figure 22, 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.

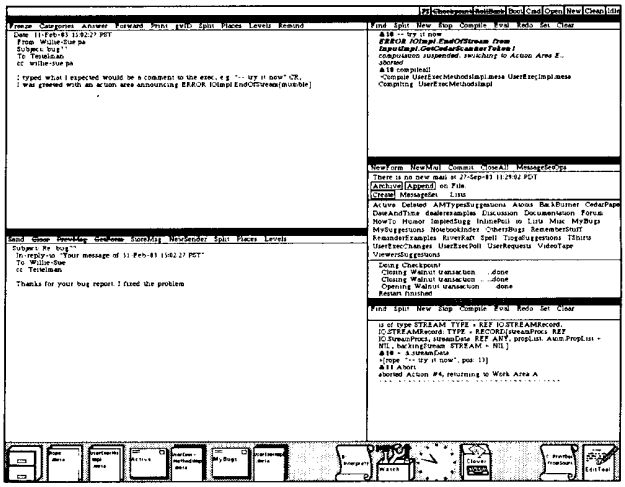

Figure 21: Cedar supports concurrency: answering mail while compiling

Meanwhile the compiler has successfully compiled the first file (notice in Figure 22 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.†47

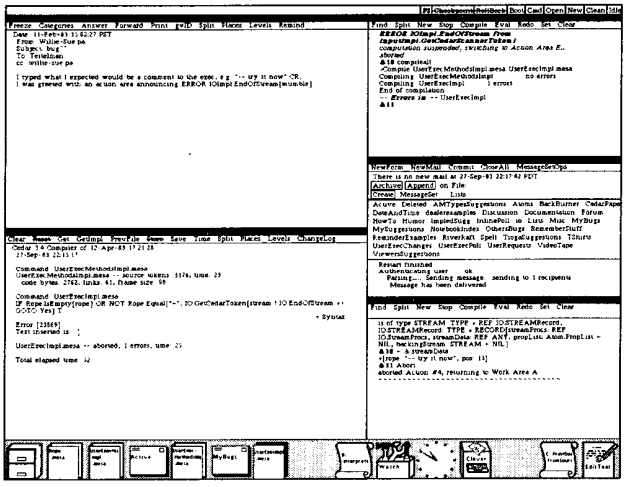

Figure 22: Compiler error log

The error is a simple syntactic error, a missing ']. 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 look at the corresponding message, which states that the user wants to be able to specify that compiler logs are always created iconic instead of always being open as in Figure 22.

†44 The Cedar language uses signals and errors as a mechanism for handling exceptional conditions. Think of a signal/error as a procedure call where the body of the procedure is determined dynamically using the call stack.

†45 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 Modelling. A preliminary version of a system modeller has been built and tested, and a more comprehensive version has been partially implemented.

†46 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.

†47 The compiler log includes for each error a position (character count) in the source file. 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. 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.

The User Profile

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.

Let's implement the new profile option. It willl allow me to demonstrate an important aspect of the Tioga Editor: its abbreviation expansion facility. I'll create a new viewer, load it with the appropriate source file, and then scroll to the place 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.

Templates as an Aid for Editing Programs

To accomplish this, I'll 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 IF ▶TEST◀ THEN ▶TRUEPART◀.†48 This template contains two fields, TEST and TRUEPART, each delimited by special brackets called *placeholders*, displayed as ▶◀. Tioga allows me to move to the next field delimited by placeholders with a single keystroke. If I am positioned at one of these fields, anything I type replaces the field. Right now, I am ready to specify the predicate for my IF-THEN statement.

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 23.†49

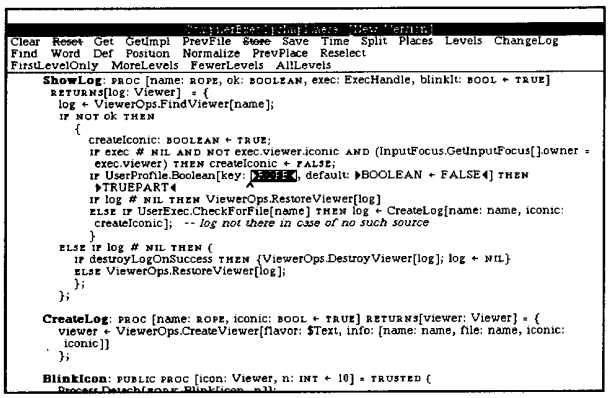

Figure 23: Computing a template for a procedure call

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 I'll make the compiler log viewer iconic. 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.

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.

Before we make this edit, let's try changing the icon for this viewer by hand, i.e., by using the interpreter.†50 So I'll reopen my interpreter Work Area, select the compiler log, and use the Eval menu button to evaluate the current selection.

```
&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.

```
&21 ← &19.icon ← typescript
typescriptt -> typescript                    †51
typescript
```

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.†52

```
&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. Let's try that.

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

Sure enough the icon for the compiler log (the icon in the center of Figure 24) is now a typescript. I can now implement this feature by editing the corresponding source.

---

†48 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.

†49 Runtime availablility of all source program information was another important item 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.

†50 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. Earlier Mesa debuggers were non-resident; 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 that shared the same address space as the programs being debugged, as well as a *complete* expression interpreter. Thus, the Cedar environment represents the first opportunity that Mesa programmers have had for realistically using an interpreter to carry out experiments.

†51 Obviously I am making some of these typing mistakes just to demonstrate the pervasiveness of the error correction facilities, but this correction is especially interesting, because 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 runtime 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.

†52 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. It is still not possible to pass types around as values, and there is no polymorphism in the language. However, at runtime, it is possible to perform a wide variety of operations on types.

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.

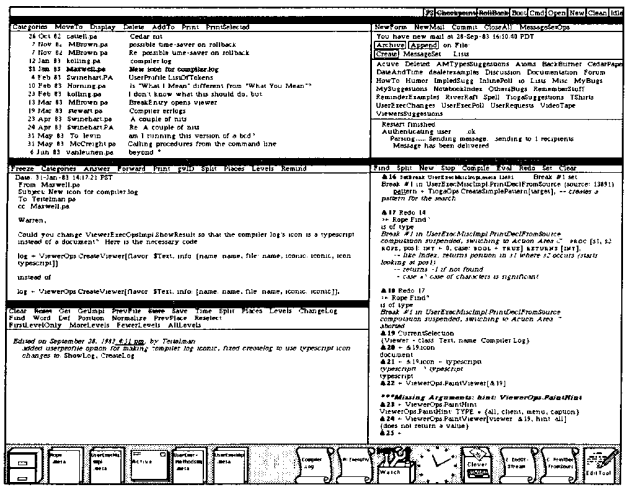

Figure 24: Now there are too many typescript icons

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 IconEditor to design a new icon for Action Areas executives (the two executives on the right in Figure 24).

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

In Figure 25, the Icon Editor shows 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).

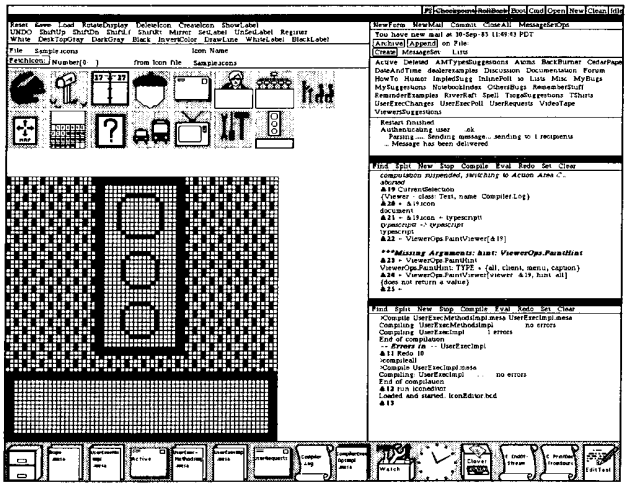

Figure 25: The Icon Editor

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. I'll make a few finishing touches to my icon – darkening the red light and adding rays of light coming from it.

I'm happy with the icon now, so I'll save it on a file and also associate the name "trafficLight" with this icon by using the Register menu button in the 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 the executives to be the trafficLight and see how it looks. First, we obtain an exec handle by 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 of: I should have said "&26.viewer.icon." Since what I did type is correct in every other respect, I can fix this by simply replaying this line, pointing at the "." in "&26.icon," and typing "viewer", as I am in the process of doing in Figure 26.†53

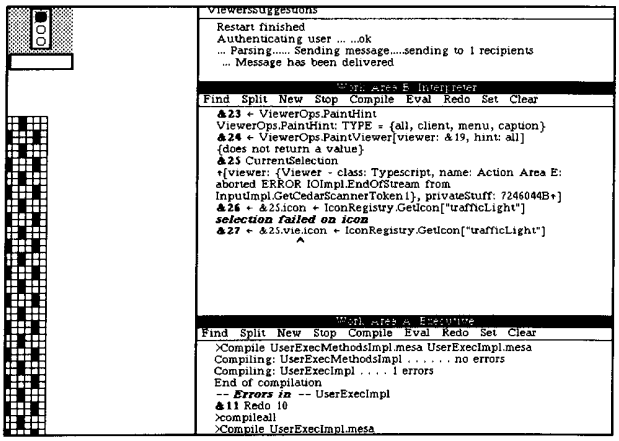

Figure 26: Editing events as they are being entered

```
&28 ← &26.viewer.icon ← IconRegistry.GetIcon["trafficLight"]
```

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.

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

And there's our traffic light (see Figure 27).

Let's go ahead and make the edit that will cause the system to use the trafficLight icon for Action Areas.

†53 In previous examples, the only editing of the typescript that we did consisted of appending characters to its end. This examples illustrates that we really can edit, in the full generality of the term, events that are being entered for execution.

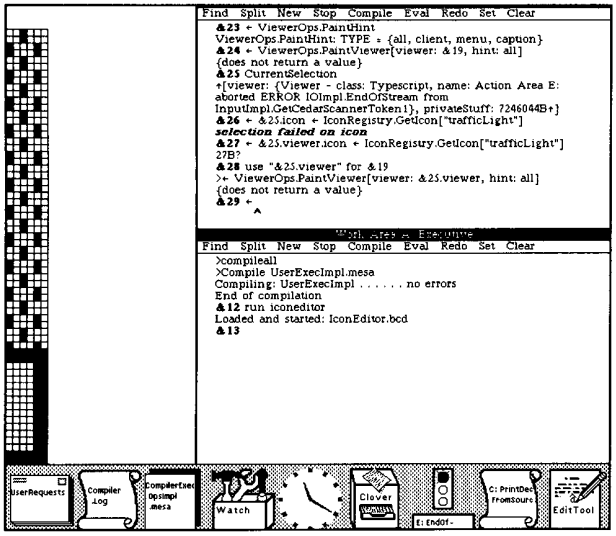

Figure 27: The Action Area icon has been changed to a traffic light

Wrapping it up

Now let's compile the rest of the files we have changed.

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

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

```
&14 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 out we have to reload the system; we can't simply replace the UserExec that is running with the new one. Reloading 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.†54

Conclusion

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 runtime, 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.

Today, in the fall of 1983, Cedar is being used by about 30 researchers. Visitors from other laboratories are envious of Cedar's emphasis on visual interaction, of its support for concurrent tasks, of its sophisticated debugging facilities, and of the wide variety of packages and tools. Cedar makes it possible for a researcher to design and implement an experimental computer system in a remarkably short period of time. For example, in the last few months, various programmers using the Cedar environment have been able to construct 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 reported favorably on how easy it was to construct real, functioning systems.

†54 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" [4]. While the changes that we made in this tour were not minor, sad to say that even had they been, we would still have been forced to reboot in order to test them out. Attacking this shortcoming is now one of our highest priority items.

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.

[2] Mark Brown, "The IO and Convert interfaces," in [5] .

[3] 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.

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

[5] J. H. Horning, editor, "The Cedar System: An Anthology of Documentation," Xerox Palo Alto Research Center Report CSL-83-14.

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

[7] 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).

[8] Butler W. Lampson, The Cedar Language Reference Manual, Xerox Palo Alto Research Center Report CSL 83-15.

[9] John Maxwell, "The Cedar Spy," in [5]

[10] Scott McGregor "The Viewers Window Package," in [5].

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

[12] Beau Sheil, "Environments for Exploratory Programming," *Datamation*, February, 1983.

[13] Warren Teitelman and Larry Masinter, "The Interlisp Programming Experience," *Computer*, April, 198

[14] Warren Teitelman, "The Cedar Programming Environment: A Midterm Report," Xerox Palo Alto Research Center Report CSL-83-11 December, 1983.

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

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