A Display Oriented Programmer's Assistant

Warren Teitelman
Xerox Palo Alto Research Center
Palo Alto. California

Abstract

This paper continues and extends previous work by the author in developing systems which provide the user with various forms of explicit and implicit assistance, and in general cooperate with the user in the development of his programs. The system described in this paper makes extensive use of a bit map display and pointing device (a mouse) to significantly enrich the user's interactions with the system, and to provide capabilities not possible with terminals that essentially emulate hard copy devices. For example, any text that is displayed on the screen can be pointed at and treated as input, exactly as though it were typed, i.e., the user can say use *this* expression or *that* value, and then simply point. The user views his programming environment through a collection *of* display windows, each of which corresponds to a different task or context. The user can manipulate the windows, or the contents of a particular window, by a combination of keyboard inputs or pointing operations. The technique of using different windows for different tasks makes it easy for the user to manage several simultaneous tasks and contexts, e.g., defining programs, testing programs, editing, asking the system for assistance, sending and receiving messages, etc and to switch back and forth between these tasks at his convenience.

Introduction

lisp systems have been used for highly interactive programming for more than a decade.+ During that period, much effort has been devoted to developing tools and techniques for providing powerful interactive support to the programmer. The Interlisp programming system [Tei4] represents one of the more successful projects aimed at developing a system which could be used by researchers in computer science for performing their day to day work, and could also serve as a testbed for introducing and evaluating new ideas and techniques for pioviding sophisticated forms of programmer assistance. Interlisp on the PDP-10 is currently used by programmers at over a do/en ARPA network sites for doing research and development on advanced artificial intelligence projects such as speech and language understanding, medical diagnosis, computer-aided msiiuction, automatic programming, etc. Implementations of Interlisp on several other machines are cuirently planned or in progress.

This paper describes a system written in Interlisp which extends the Inierlisp user facilities to take advantage of a display.tt The paper is not an "idea" paper in the sense that Artificial Intelligence papers usuallv are. Instead, this papcr describe', a woiking sysiein which implements and *inttrratcs* a number of idvis and techniques previously reported in The liteiature by several dillYicnl individuals, including the author. The idea of a display composed of multiple, overlapping regions called "windows" is attributable to and an essential pail of the Smallt.dk programming system designed and implemented by the learning Research Group at Xerox Reseaich (enter [1 RG]. In particular, much of the way that windows are used in the system described here was influenced by the work of Dan Ingalls on the Smalltalk user interface. The idea of using the display as a means for allowing the user to retain comprehension of complex program environments, and to monitor several simultaneous tasks, can be found in the work of Dan Swinehart [Swi], The use of the "mouse" as a pointing device for selecting portions of a display goes back to the early work on NFS [Eng]. Finally, the techniques used for automatic error correction and the idea of having the user interact with the system through an active intermediary winch maintains a history of his session, both *of* which appear in this paper, are parts of the standard Interlisp system [Tei1][Tei2]. The work reported in this paper is of interest primarily in how the realization of these various ideas in a single, integrated, working system dramatically confirms their value.fit

Overview of the System

The system described in this paper is implemented on a version of Interlisp [Tei4] running on MAXC, a computer at the Xerox Research Center in Palo Alto. This computer emulates a PDP-11), and runs the Tencx operating system, so that from the standpoint of the user, the system he is using is Interlisp-10. The raster-scan display used by the system described in this paper is maintained by a separate 65K 16 bit word mini-computer. The minicomputer is linked to MAXC through an internal network, and implements a graphics protocol similar to the Network Graphics Protocol [Spr]. but specialized for text and raster-scan images. All of the work described in this paper deals with the "high end" of the system, i.e., the user interface, and is wutten entirely in Interlisp.

The user communicates with the system using a standard typewriter-like keyboard. In addition, he has available a pointing device commonly called a "mouse" [Eng] used for pointing at particular locations *on* the screen. For those unfamiliar with this device, the mouse is a small object (about 3" by 2" by 1") with three buttons on its top. The system gives the user continuous feedback as to where it thinks the mouse is pointing by displaying a cursor on the screen. The user slides the mouse around on his working surface (causing bearings of wheels on the bottom *of* the mouse to rotate), and the system moves the cursor on the

f An excellent survey of the state of the art may be found in [San]

t The author would like to acknowledge and thank R. F. Sproull and J Strother Moore, who designed and implemented critical support facilities without which this system would not have been possible, and whose ideas and intuitions provided extremely valuable guidance and inspiration during the development of the system. "I he form and capabilities of some of the display primitives in the current system were suggested by an earlier version of a display text facility for Interlisp designed by Terry Winograd. Finally, all of the work described herein depends heavily on th" leverage provided by the Interlisp system itself, which is the result of the efforts of man> individuals over a period of almost a decade, made possible by continuing ARPA support over that period.

!JtWhen I first began woik in 1969 on what was to become DWIM. the automatic error correction facility of Interlisp, by implementing a primitive spelling corrector which would automatically correct a certain class of user spelling errors, I discussed this project at length with a colleague over a period of months. One d:is soon after this facility was finally completed and installed in our f isp system, this same colleague rushed to my office and in i*rral excitement exclaimed that the system had corrected an error. I was surprised at his enthusiasm, since we had been discussing this system tor months. He replied, "Yes, but it really did it!" the system described herein implements ideas that many of us have long been sayinu would be a good thing to have. And they really are!

display. The user indicates that the mouse has arrived at some desired location by pressing one of the three buttons on the top of the mouse. The interpretation of the buttons depends on the particular program listening to the mouse. For example, when the mouse is positioned over a piece of text, and one of its buttons pressed, the corresponding text is "selected." Such selections are indicated by inverting the text, i.e., displaying it as white characters on a black background.

The user interacts with the system either by typing on the keyboard, or by pointing at commands or expressions on the screen, or an asynchronous mixture of the two. In particular, any material that is displayed on the screen can be selected and then treated as thought it were input, i.e., typed.

The ability to be able to select, i.e., point at, material currently displayed and cause it to be treated as input is extremely useful and situations where such a facility can be used occur very often during the course of an interactive session.

Why is such a facility useful? Because most interactions with a prus.',raminim; system are not independent, i.e., each "event" bears some relationship to what transpired before, usually to a fairly recent event. Being able to point at (purlions *o()* these events effectively gives the user the power of *pronoun reference,* i.e.. the user can say use *this* expression or *that* value, and then simply point. This drastically reduces the amount of typing the user has to do in many situations, and results in a considerable increase in the effective "bandwidth" of the user's communication with his programming environment.

The user views his environment through a display consisting of several rectangular display "windows". Windows can be, and ficquenlly are, overlapped on the screen. In this case, windows that are "underneath" can be brought up on top and vice versa. The resulting configurauon considerably increases the user's effective working space, and also contributes to the illusion that the user is viewing a desk top containing a number of sheets of paper winch the user can manipulate in various ways.

One facility provided by these windows that is not available with sheets of paper is the ability to *scroll* the window forward or backward to view material previously, but not currently, visible in the window. Thus a single window can be used to view and manipulate a body of text that would icquire many sheets of paper.

bach window corresponds to a different task or aspect of the user's environment. for example, there is a tYIMSCRirr window, which contains the transcript of the user's interactions with the Lisp interpreter through the programmer's assistant, a WORK AREA window winch is used for editing and prettyprmting, a HISTORY. window, a BACKTRACF window, a MESSAGE: window, etc. Using different windows for different tasks

...makes it easy for the user to manage several simultaneous tasks and contexts, switching back anil forth between them at his convenience.

Being able to switch back and forth between tasks icsults in a relaxed and easy style of operating more similar to the way people lend to work in the absence of restrictions. To use a programming metaphor, people operate somewhat like a collection of coroutines corresponding to tasks in various states of completion. These coroutines are continually being activated by internally and externally generated interrupts, and then suspended when higher priority interrupts arrive, e.g., a phone call that interrupts a meeting, a quick question by a colleague that interrupts a phone call, etc. Our previous experience with Interiisp supports the contention that it is of great value to the user to he able to switch back and forth quickly between related tasks. 1 he system described in this paper makes this especially convenient, as is illustrated in the sample se.sMon presented in the body of the paper.

One technique hcavib cmplovcd thioughoni the system is the use ol *menus.* A menu is a lypc of window that causes a specified operation to be p».; IOIHHM! when a selection made in that window. Menus scive a numhci of impottant functions. \ hey make it easy for the ti.ser to specify an operation without having (o t\ne. Thev act as a prompt for the ircr by providing him with a repeiloire of commands from which to choose, tor example, often a user will not remember the name of a command, or may not even be aware of the existence ol' a command.

However, most importantly, *menus llv facilitate context switching.* As with most systems, the interpretation of the user's keystrokes (with the exception *oi* interrupt characters which usually have a globallv detmed effect) depends on the state *i)*\ the system, f or example, when addressing the Lisp interpreter, the characters that the user types are used to construct Lisp expressions which are then evaluated. When using the editor, the characters are inserted in the indicated expression, etc. 1 he important point is that once the user starts typing, he normally has to complete the operation or abort it. However, by selecting a menu command using the mouse, even in the midst of typing, the user can temporarily suspend the operation he is pel forming, go off and do something else, and then return and continue with his current context. This is also illustrated in the sample session below.

A Samojr Session vvitji_tlie System

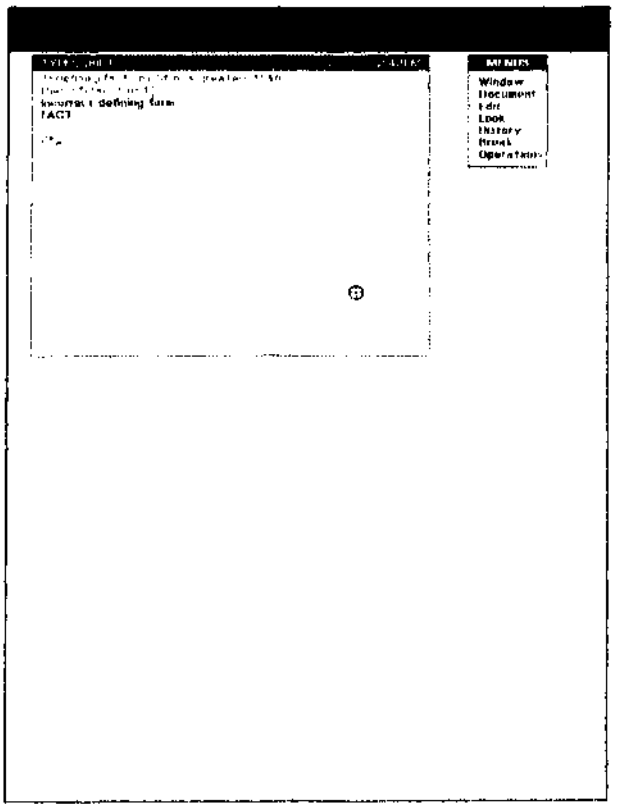
Since so much of the utility o\' the system desciibed in this paper rest on visual effects, it is difficult to transmit the feci and smoothness of the system through words. Therefore, the form chosen for presenting, the system in this paper is to take the leader through a sample session with the system, using frequent "snapshots" of the display as a substitute for the actual display itself. This session is divided into two parts. The first part is a "toy" session, in that the user is not performing any serious work. It is included only to introduce the salient features of the system. The second part of the session shows some more sophisticated use of these features in the context of an actual working session involving finding and fixing bugs, testing programs, sending and receiving messages, etc.

For readers not familiar with I isp, please ignore Lisp related details (which we have tried to minimi/e). I he impoitant point is the way tic system allows the user to switch back and forth between several tasks and contexts. Such a facility would be useful in any piogramining environment.

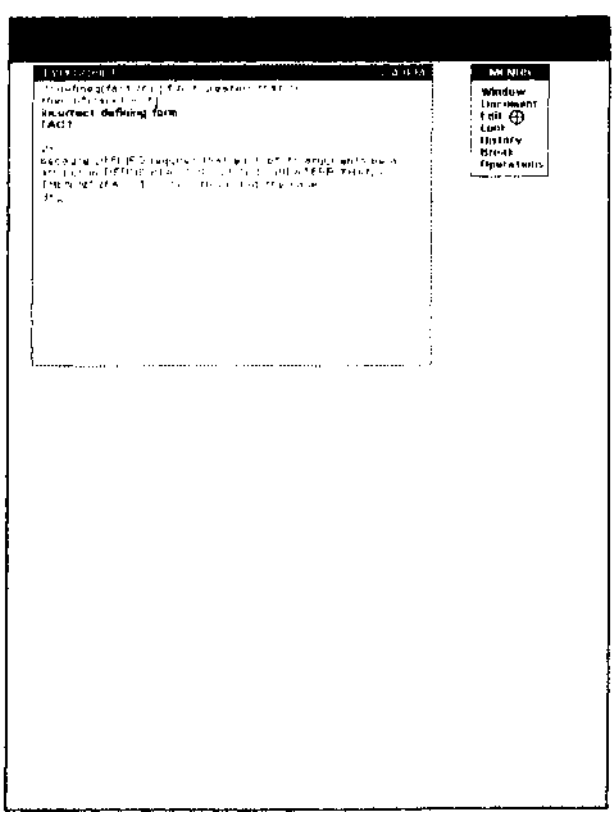
Sample_Session _ - Part 1

1. Figure 1 shows the initial configuration of the screen. Three windows are displayed: the TYPESCRIPT window, which records the user's interactions with the programmer's assistant and the Lisp interpreter; the PROMPT window, which is the black region without a caption at the top of the screen used for prompting the user; and a *menu,* which is the smaller window with caption MENUS to the right *of* the IYPESCRIPT window.j A menu is just like any other window, except that whenever a selection is made in a menu, a specified operation is also performed. This particular menu is a menu of *menus,* hence its caption. If the user selects one of its commands, each of which is the name of a menu, the corresponding menu will be displayed at the location he indicates. He can then select, and thereby perform, commands on that menu. The crosshairs shape in the lower right hand portion of the TYPESCRIPT window is the *cursor,* and indicates the current position of the mouse.

In Figure 1, I have just typed in a Lisp definition for the function FACT (factorial). Fisp has given me the error message "incorrect defining form" (displayed in bold face to set it off). The system displays a blinking caretff to indicate where the next character that I type, or the system prints, will be displayed. In Figure I, the caret now appears immediately following the *"2<-",* where 2 is the event number for my next interaction with the programmer's assistant, and <- is the "ready" character.



2. I don't understand what caused this error, so I type ? to the p.a. (programmer's assistant), requesting it. to supply additional explanatory information. The p.a. looks at the previous event to determine the nature of the error. In this case. tram: built-in information about the arguments to DI I INI■«..», the p.a. tells me that the problem is that DI.HNR; encount'-i'd an atom where it expected a list, i.e., a left parentheses is missing from in front of the word "fact".!" | j Since the programmer's assistant is maintaining a history of mv interactions with the system. I don't have to retype the I>! IINI o expression. Instead, I can edit what I have already typed, and simply insert the missing left parenthesis. The ! i>ii menu will allow me to perform various editing opnations using, the mouse for pointing, and the keyboard, where necessary, for supplying text. In figure 2, I have alicady moved the mouse so that the cursor is positioned over the EDIT command on the MhNiis menu, in preparation for "bringing up" the roil menu.



I"! IK- "plaid" effeet of the hackground in the figures is an artifact of the pmdmtion/iedn-hon process. I he aelu.tl background wit (he display ts a uniform grey.

If In these figures, (he tare! is always shown in ils "on" position.

fi'TIf Ihe p.a. did not know anything about this particular error, il would refer to the index of Ihe on-line Inlerlisp Reference Manaial and present the toi respondmp text as:.ociated with the crro< message by way of explanation. I he user can also airmenl the bnill-in mini malum that the p.a has about syacm functions by mfoinnnj', the p.a. about the requirements of his own functions. He ean then use the ? command to explain errois in his own programs.

3. I press a button on the mouse to select the EDIT command in the MENUS menu. The system indicates the selection by displaying EDIT as white on black. The PROMPT window tells me to use the left button on the mouse to indicate where I want the center of the (EDIT) menu to appear. The cursor is changed to an icon of a menu with a cross in its center to suggest the operation that is pending. At this point, I don't *have* to complete this operation. I can type in other expressions to the programmer's assistant, perform other menu operations, etc. The process which is waiting for me to supply the indicated information is simply a co-routine which has been suspended.! However, since I want to fix up the DFEINEQ expression before going on to anything else. I move the cursor to the position at which I want the EDIT menu to appear, which is below the MENUS menu and to the right of the TYPESCRIPT window, as shown in Figure 3.

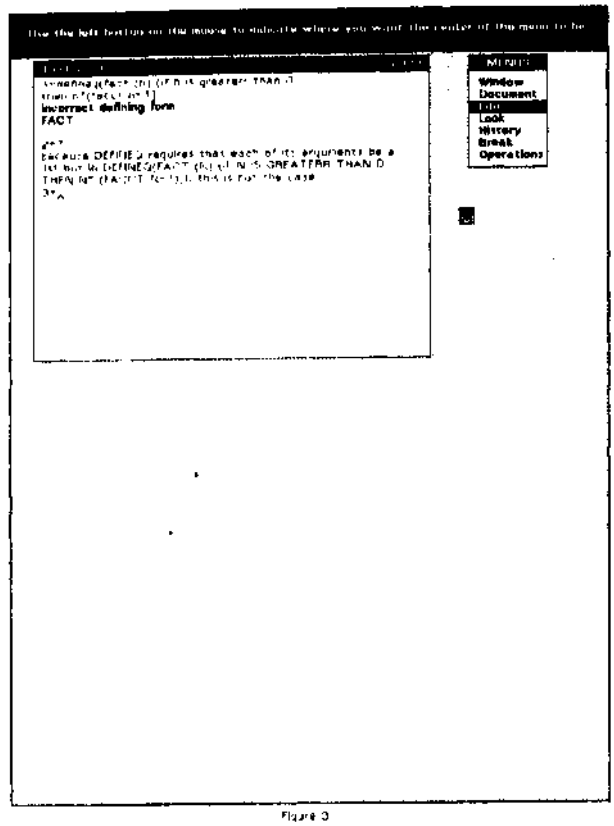


Figure 3

4. I press the left button on the mouse, causing the EDIT menu to appear at the location of the cursor. In this position, the EDIT menu slightly overlaps both the TYPESCRIPT window and the MENUS menu, so the system automatically adjusts the EDIT menu by sliding it off these windows to its location as shown in Figure 4.|f

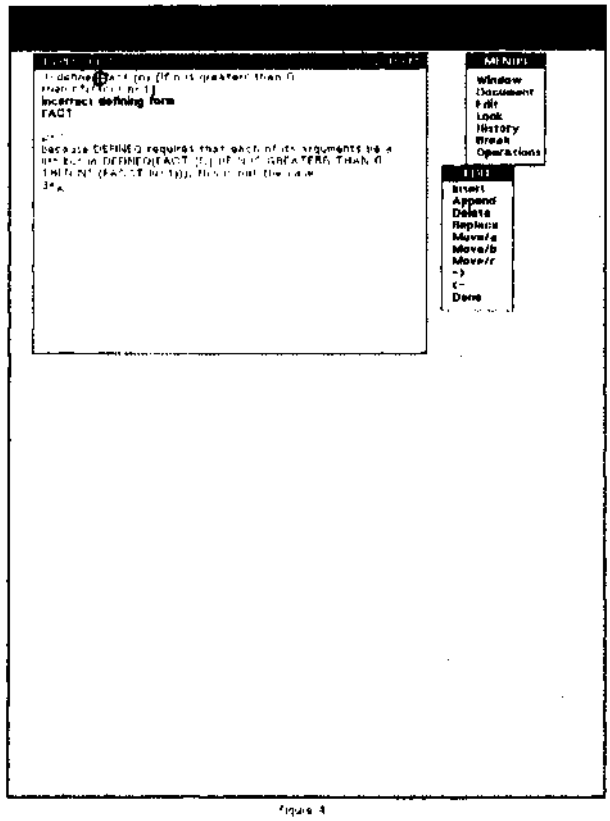


Figure 4

'I'See description of the "Spaghetti Stack" facility in [Eob] and [Tei4].

I t ' could force the EDIT menu to overlap the TYPf SCRIPT window by positioning it exactly using one of the commands on the WINDOW menu. However, since in this case I only positioned the menu approximately, the system tries to "Do What I Mean", a philosophy of system design we have tried to follow throughout the Interlisp system

5. Now I am ready to edit. I select the left parenthesis in the first line of the TYPESCRIPT window, and then select the INSERT command on the EDIT menu. The line of text in the TYPESCRIPT window is broken just before the selection (the left parenthesis), and the caret is moved to that location. The PROMPT window instructs me to input material. Anything I type will appear at the location indicated by the caret.

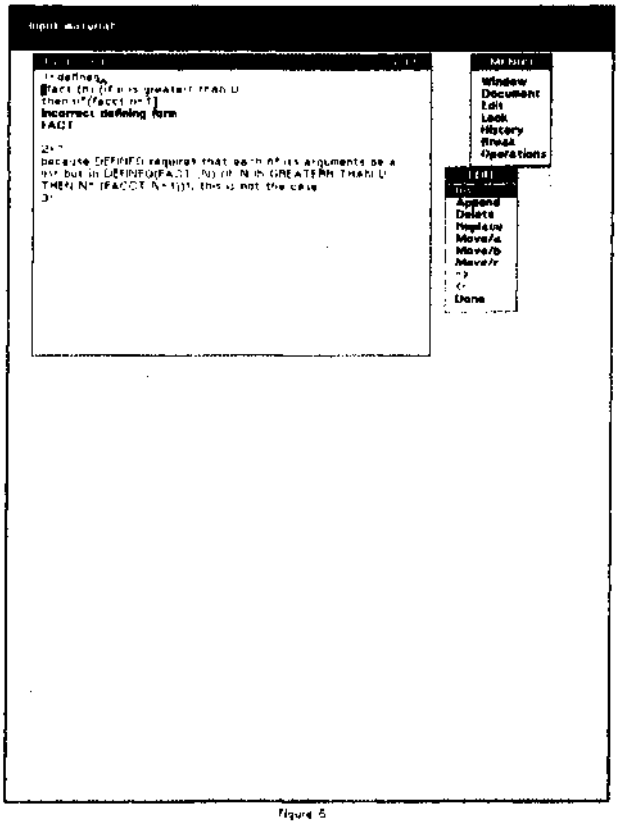


Figure 5

6. I type in a single left parenthesis, and terminate the INSERT operation. The line of text I have been editing is rejoined, and the caret returned to the appropriate location at the end of the TYPESCRIPT window. I now wain to cause the corrected text to be *re-input* in order to perform my original operation, i.e., define my function. Therefore, I select the text by first selecting the "d" in "iltTineq" and then extending this selection through the final "]". Then, using the same method as previously shown for bringing up the EDIT menu, I bring up the WINDOW menu in order to obtain the command for inputting selected material.

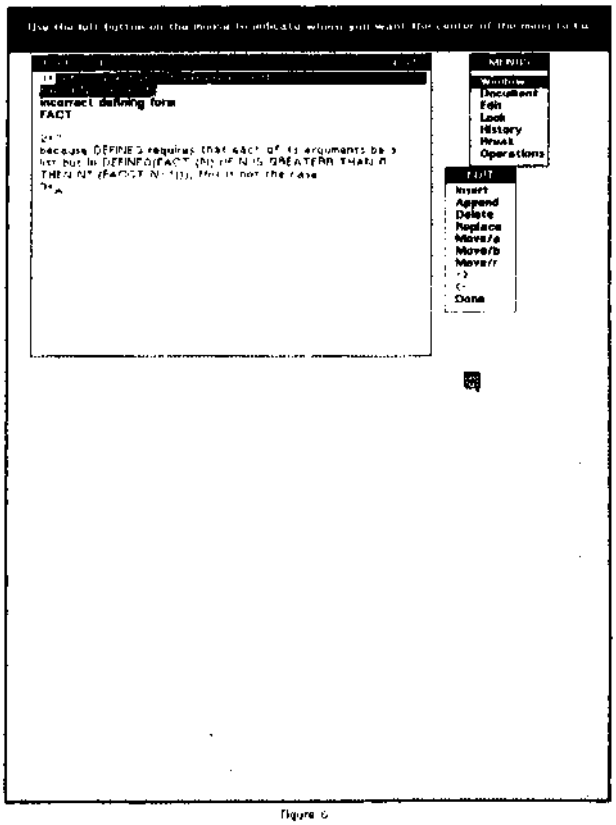


Figure 6

11. I still don't understand why the error occurred, so I try typing the ? command again. In this case, the programmer's assistant tells me that the problem is that one of the operands to * (the MULTIPLY operator) was (FACT N-1) and that the value of (FACT N-D is NIL when N=1. In other words, when FACT is called with N=o, it returns NIL. The p.a. is able to generate this explanation because (1) it knows that all of the arguments to * must be numbers, and (2) it can examine the state of the computation on the stack. In this case, it found that the second operand to iTIMts was NIL, which is not a number, and that the expression that produced this particular value was (FACT N-1) in the expression (N.(FACTN-1)) which is contained in the function FACT, and that at the time this call occurred, the value of N was 1.

I now realize that the problem is simply that I neglected to specify the value of FACT for N=o.f Therefore, I prettyprint the definition of FACT in preparation for editing it. Figure II shows the definition of FACT prettyprinted in my WORK AREA window, which automatically appeared when prettyprint was called. Note that the definition of FACT now shows the two misspelled words, GREATERR and FACCT, spelled correctly.

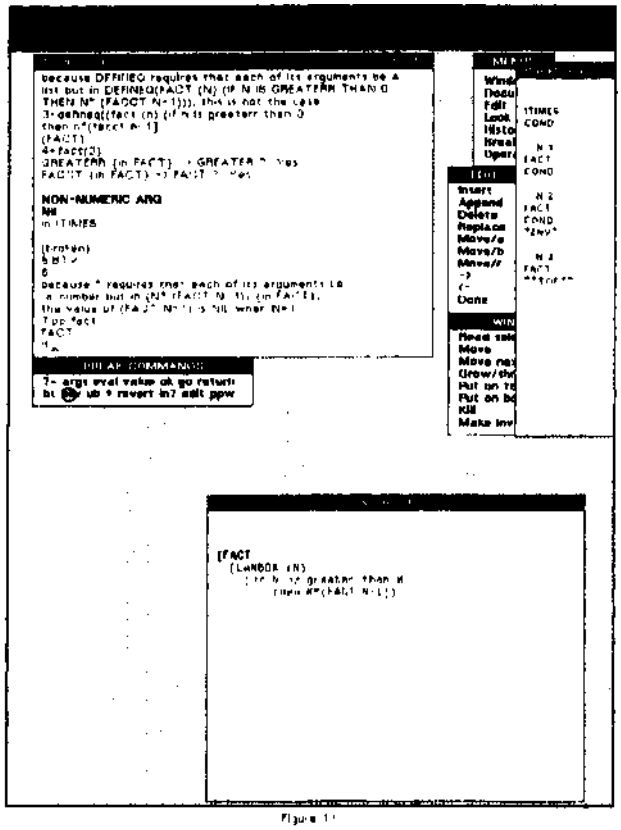


Figure 1'

12. I select the right square bracket in the definition of FACT in the WORK ARIA window, and then select the INSERT comand on the EDIT menu. The EDIT menu automatically moves so as to be close to the window that I am editing. I make the necessary correction by typing ") ELSE I", i.e. if N is not greater than 0, FACT should return 1. Figure 12 shows the display just before I complete the INSERT. Note that the caret appears in the WORK AREA window where I am typing. The cursor is in the upper right hand portion of the screen at the location of the INSERT command before the EDIT menu moved to be close to the WORK AREA.

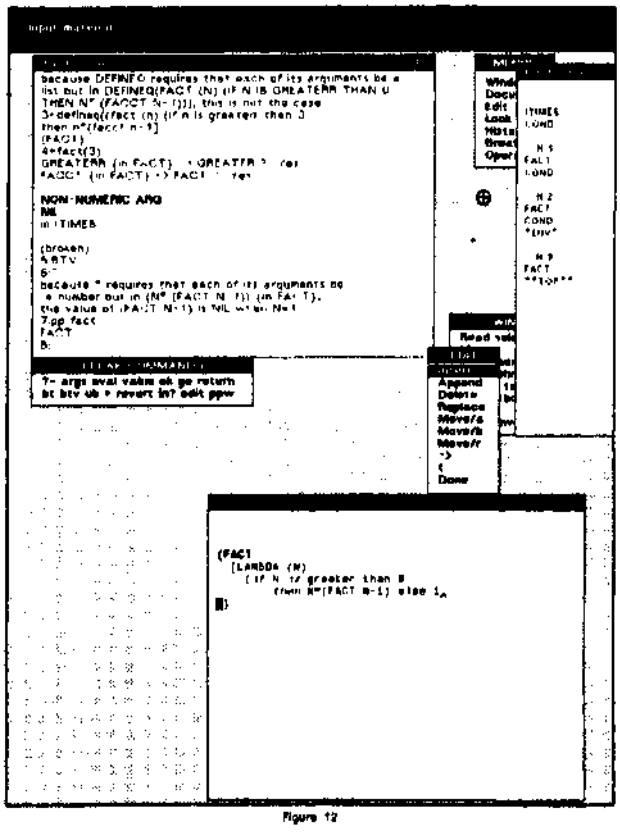


Figure 12

f i n Interlisp, *if* none of the predicates of ;tn if-then expression evaluate true, the value of the expression defaults to NIL.

13. I complete the INSERT, and then select the DONE command on the EDIT menu to indicate that I am finished editing this expression. The PROMPT window reports that the definition of FACT has been changed. Note that I did not *have* to finish editing FACT at this point: I could have typed in expressions to be evaluated, performed other menu operations, etc.. even edited other expressions, before selecting the DONE command for this expression. This is another example of being able to suspend different tasks in varying states of completion and go back to them at some later point.

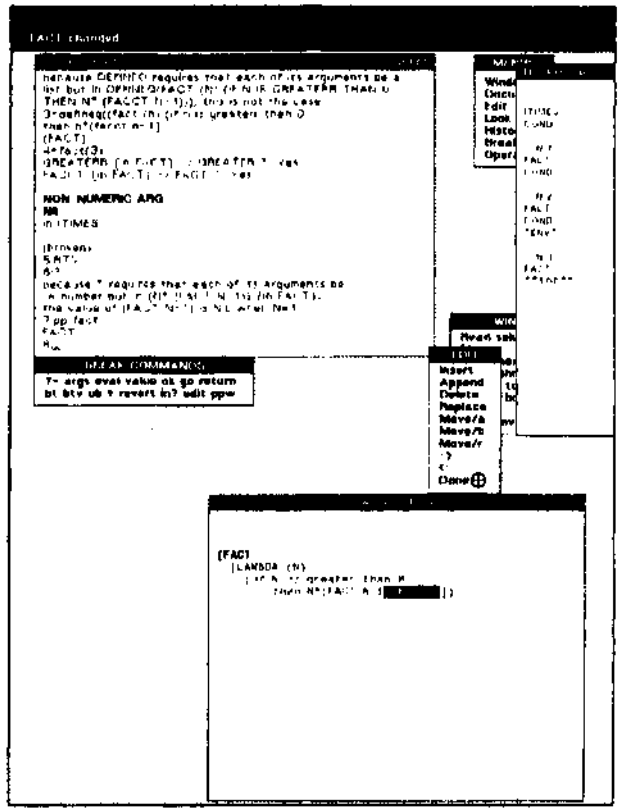


Figure 13

14. I now test out my change by typing fact(2), which works correctly. Now I want to *continue* with the computation. Note that I am still in the original break that followed the error. The arithmetic operation * (i.e., the Lisp function ITIMHS) is still waiting for a number to be used as a multiplicand. I therefore select the RETURN command on the BREAK menu. The PROMPT window tells me to INPUT EXPRESSION and the caret moves to the PROMPT window. I type 1 as the value to be returned from this error break. Figure 14 shows the display at this point just after I type 1, which is echoed (displayed) in the PROMPT window.

Note: in actual practice, for a computation as trivial as FACT(3), I would probably simply reset (abort back to the top) and reexecute FACT(3) rather than bothering to continue the computation, since so little has been invested in getting to this point. However,

being able to continue a computation following; an error is especially useful when an error occurs following a significant amount of compulation, or when the computation has left things in an "unclean state" as a result of global side effects. Such a facility is also essential for good interactive debugging.

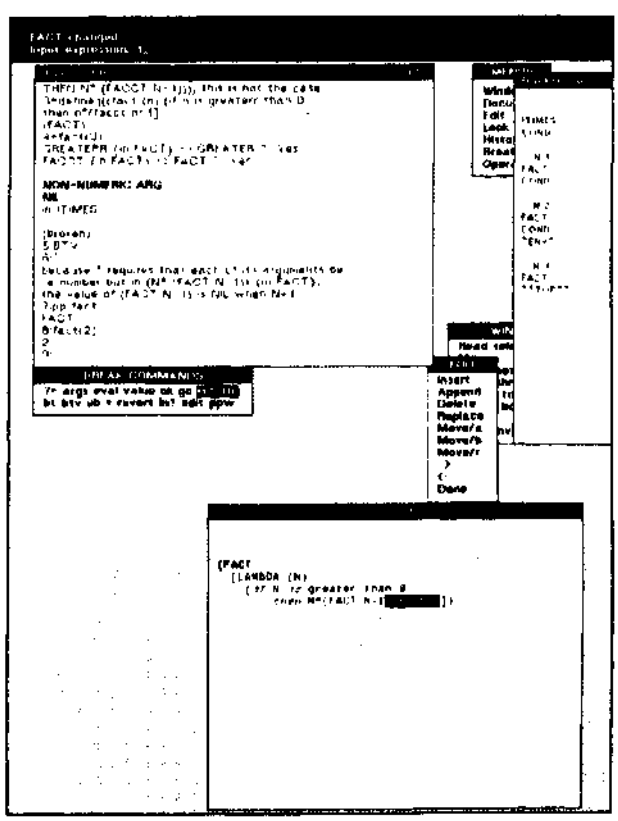


Figure 14

15. I complete typing the expression for the RETURN command, thereby causing 1 to be returned as the value of the break, which causes (1 * I) to be computed and returned as the value of FACTO), which then causes (2 * 1) to be computed, etc., and finally the original computation of FACTO) finishes and returns 6 as its value as shown in Figure 15. in the next to the bottom line of the TYPESCRIPT window.

I now want to try FACT on some other values, so I bring up the HISTORY menu, and select the usr. command, which is a command to the programmer's assistant to reexecute a previous event, or events, with new values. The PROMPT window instructs me to select the targets and to input the objects to be substituted. I select the "3" in FACTO) (near the lop of the TYPESCRIPT window) and input "4 5 10" (echoed in the PROMPT window), i.e., I am requesting that FACT(4), FAC'IO) and FACT(10) be computed.

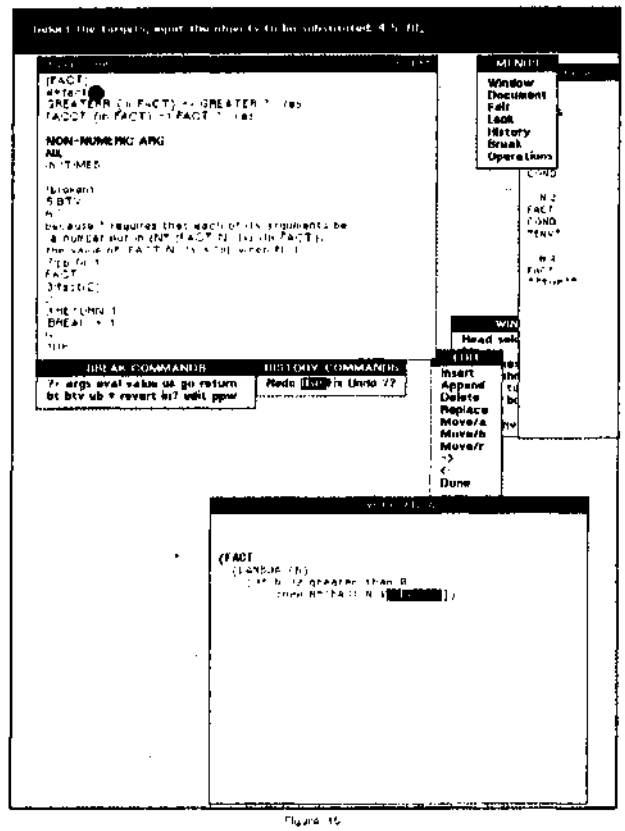


Figure 15

16. The resulting history operation is equivalent to typing USF 4 5 in FOR 3 IN <14' which the p.a. prints in the TYPFSCRIPT window to show me what is happening. This USF command now causes three computations to be performed, corresponding io the result of substituting 4 for 3 in FACT(3). the resuit of substituting 5 for 3 in FACTO), and the result of substituting JO for 3 in FACT(3). The values produced by these three compulations, 24, 120. and 3678800, are printed in the TYPI-NCKUM' window, as shown in Figure 16. Finally, I ask for a replay of the history of my session, by selecting the ?? command in the 11isIORY menu. The HIMOKY window is brought up, and the history of my session, in reverse chronological order, is printed in this window, as shown in figure 16.ff

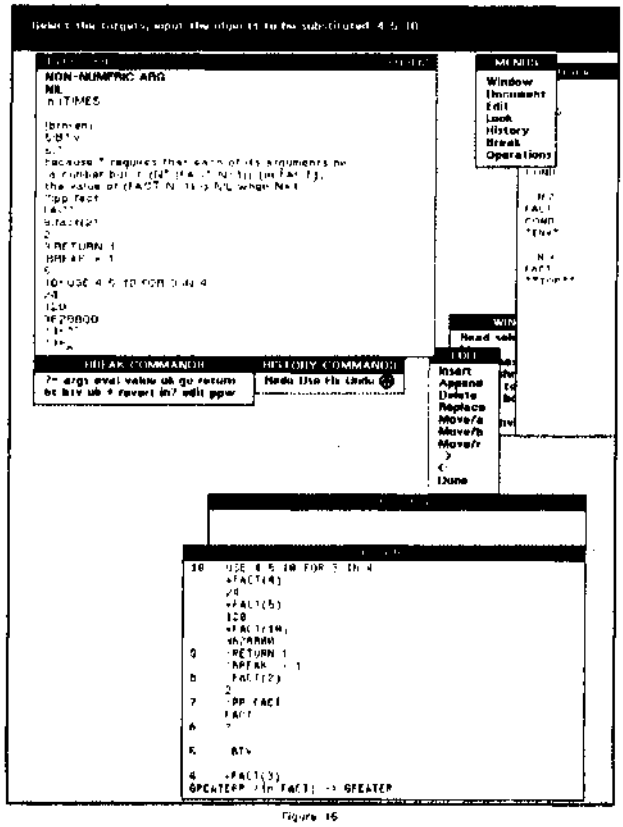


Figure 16

f4 is the event number of the .-vein corresponding k> FACTO).

ft In addition io seeing a replav of his history, the user can also *scroll* the (contents of the) 'IYPFSCKIPJ WINDOW backward, m time to we the transcript of carlin interactions with the system. The difference between the history anil the IYPFSCRIPT is that the TYPI SCRIPT contains a record of all chaiailcrs input or output, e.g., includes messages printed by the system and by the user's prograions The histoiy contains a subset of these duracteis. orram/cd according' to events. For cample, 6, the value returned by FACTO), actually appears IX lines below FACTO) in the IMTSCRIPI window, but in the HISTORY window, it would be shown as the value of event number *A,* regardless of the fact thai events 5 thru 9 occurred between the lime that event 4 was begun and the time it completed. .

This completes the "toy" session designed to illustrate some of the basic features of the system. Note that at this point the display contains nine different windows. Five of these windows are control windows (menus). The other four windows describe various processes. Note that the windows have not been a burden on the user: he does not "manage" the windows, although he could perform explicit operations on them such as changing their position, or size, or shape, or editing their contents as we have seen. The feeling to the user is that the windows more or less manage themselves, and this contributes greatly to the smoothness of the system.f

Conclusions

The system decribed in this paper has been in use by actual users other than the author only a few months. However, our conjectures about the usefulness of this kind of facility were if anything conservative. The ability to suspend an operation, perform other operations, and then return without loss of context is widely appreciated. The technique of using different windows for different tasks docs make this switching of contexts easy and painless.

liven when the user is not switching contexts, the use of multiple windows is extremely helpful. For example, a standard complaint with conventional display terminals is that material that the user wants to refer to repeatedly, e.g., a printout of some function, or a record of some complicated interaction, is displaced by subsequent, incidental interactions with the system. In this situation when using a hard copy terminal, the user simply tears off the portion he is interested in and saves it beside his keyboard. Being able to freeze a portion of the user's interactions in a separate window, such as the WORK AREA. while allowing subsequent interactions to scroll off the screen seems to combine some of the best aspects of hardcopy and display terminals.

Finally, users just seem to enjoy aesthetically the style of interacting with the system, such as using menus, the feedback via the prompt window and changing cursors, being able to scroll the windows back and forth, etc. We think this is an area that will see an increasing amount of activity in the future as the cost of bit map displays and the necessary computing power to maintain them continues to drop.

REFERENCES

[Bob] Bobiow, D. G.. and Wegbreit B., "A Model and Stack Implementation for Multiple Environments," *Communications of the ACM,* Vol. 16, 10 October 1973.

[Eng] English, W. K., Engelbart, D. C, and Herman, M. I.., "Display Selection Techniques for Text Manipulation," *IEEE Transactions on Human Factors in Electronics,* Vol. HFE-8, No. 1. March 1967.

[LRG] Learning Research Group. *Personal Dynamic Media,* Xerox Palo Alto Research Center, 1976. Excerpts published in *IEEE Computer Magazine,* March 1977.

[San] Sandewall. E, "Programming in an Interactive Environment: The Lisp Experience," Matematiska Institutionen, University of Linkoping, Sweden. (to be published in *CACM).*

[Spr] Sproull. R. F., and Thomas, E. L., "A Network Graphics Protocol," *Computer Graphics, SIUGRAPH Quarterly,* Fall 1974.

[Swi] Swinehart, D. C, "Copilot: A Multiple Process Approach to Interactive Programming Systems," Stanford Artificial Intelligence Laboratory Memo AIM-230, Stanford University, July 1974.

[Teil] Teitelman, W. "Toward a Programming Laboratory," in Walker, D. (ed.) *International Joint Conference on Artificial Intelligence,* May 1969.

[Tei2] Teitelman. W. "Automated Programmering - The Programmer's Assistant," *Proceedings of the Fall Joint Computer Conference,* December 1972.

[Tei3] Teitelman. W. "CLISP - Conversational Lisp," *Third International Joint Conference on Artificial Intelligence,* August 1973.

[Tei4] Teitelman, W. et al., *Interlisp Reference Manual,* Dec. 1975, Xerox Palo Alto Research Center.

'Thc second part of the session, which shows more sophisticated use of the above features in the context of an actual working session involving finding and fixing bugs, testing programs, sending and receiving messages, etc.. may be found in an expanded version of this paper available as a Xerox CSI Report, *A Display Oriented Programmer's Assistant,* by Warren Teitelman.