Symbolics Software

This document gives an overview of the software provided with the Symbolics LM-2 and 3600 Symbol Processing Systems.

Symbolics, Inc.
9600 De Soto Avenue
Chatsworth, California 91311
(213) 998-3600

This document gives an overview of what software is provided by Symbolics, Inc. for use with its LM-2 and 3600 Symbol Processors. Hardware is not discussed herein; all Symbolics computers run this software. This software forms an integrated and extremely powerful program development environment. The systems programming language for the Symbolics machines is the Zetalisp dialect of Lisp. In this writeup, we first discuss the user interface to the Symbolics system: what you see as a user. This includes the window system, the file system, the text editor, the mail reader, and other interactive systems and tools. Then, we discuss the Zetalisp programming environment: the language itself and various tools and features that make programming easier.

All of the software described below has been designed, written, tested, and installed for a user community (except as noted). What you are about to read is not what we plan to do in the future; it is what has already been completed. Almost all of this software has been installed at the Massachusetts Institute of Technology and is running on over twenty Lisp Machines owned by the Artificial Intelligence Laboratory, the Laboratory for Computer Science, and the Department of Electrical Engineering and Computer Science. It has been field-tested by a large community of M.I.T. student, staff, and faculty, over a period of more than five years. These users have been enthusiastic but demanding, and their trouble reports and suggested improvements have been continually integrated into the software system. As a result, this software is well-tested and mature. Most of the writers and maintainers of this software are now employees of Symbolics, Inc., and they continue to maintain and improve the software system. All software is sold with complete sources, and maintainance for one year.

Printed on the Symbolics LGP-1 Laser Graphics Printer

The User Interface

Text Editor: Zmacs

The text editor of Symbolics's system is called Zmacs. It is a real-time display-oriented text editor; this means that the text you are editing is always visible in front of you, and your commands are executed as you issue them, so that the effects of your commands are immediately visible. Zmacs can be used for editing text or programs. It has specialized features for editing programs in the Zetalisp language and for communicating with the Zetalisp environment. It takes advantage of the graphical input and output capabilities of the terminal to provide greater ease of use. It is easy for a beginner to learn, and offers many sophisticated commands for the advanced user. You can customize its behavior and add extensions.

Over the last four years, the Emacs text editor for the PDP-10 has become extremely popular throughout universities and the A.I. community. Editors featuring the basic Emacs command set are now available on many timesharing systems. Zmacs uses this command set; in fact, most Emacs commands are implemented compatibly in Zmacs. In a computation facility that includes time-sharing systems that run Emacs-like editors, users will be able to move back and forth between the Symbolics system and time-sharing systems without having to learn two different editors.

Basic commands in Zmacs, like Emacs, are modeless. To insert text at the current cursor position, you just type the text; there is no "insert mode". To move the cursor around, you can use simple single character commands to move it forward or backward, or to the next or previous line. Zmacs is easy to learn; with about fifteen basic commands, you can edit text effectively. As you gain proficiency with Zmacs, you can start learning about more powerful commands. For example, there are commands to move around in the text in units of lines, words, sentences, and paragraphs. You can mark a section of text and move it or copy it somewhere else in the file. You can supervise the replacement of all occurrences of a text string with a second text string; for every occurrence of the first string, Zmacs shows you the occurrence in its context, and you can decide whether or not it should be replaced by the second string. You can indent whole regions of text by arbitrary amounts. You can perform text filling or justification on paragraphs or arbitrary regions. You can deal with many files at the same time, switching between them. You can also have any number of windows on the screen at a time, each showing different files being edited, or different parts of the same file. Commands that are frequently used can be issued with only one or two keystrokes, for speed; commands that are less frequently used have longer, mnemonic names; you type these names with the assistance of sophisticated command completion. Commands have built-in documentation; you can immediately get help for any command in Zmacs.

Zmacs also has extensive features for editing of Zetalisp programs. There are commands to move back and forth over Zetalisp expressions; Zmacs knows how to match up parentheses with each other, and it completely understands the syntax of Zetalisp so that it knows, for example, to ignore parentheses that are inside character strings or otherwise "quoted". While it is difficult to create unbalanced expressions with this sort of sophisticated assistance, should one occur you can use a command called "Find Unbalanced Parentheses" that looks over an entire file and positions you at sites of suspected parenthesis errors. Zmacs also knows stylistic rules for indentation of Zetalisp programs, and has a command that inserts the right amount of indentation on a line of text. When you type in a Zetalisp program to Zmacs, as you get to the end of a line of text, you can type the Return key to simply insert a carriage return, but alternatively you can type the Line key to insert a carriage return and the appropriate amount of indentation for the new line of the program. If you have made a syntax error in previous lines that leaves parentheses incorrectly matched, you can immediately tell because the automatic indentation doesn't go where you expected. You can also run this automatic indentation over an entire Lisp function or the whole file, and it will reindent each line correctly. Whenever the cursor is just to the right of a close parenthesis (for example, just after you insert a close parenthesis), the matching open parenthesis blinks on and off, providing instant confirmation of correct parenthesis balancing.

Zmacs also interacts with the Lisp environment. If you type in a new Lisp function or modify an existing one, you can evaluate the function definition, or compile the function, with a single keystroke. You can do some editing on various parts of a Lisp program, and Zmacs will automatically find all of the functions you have modified, and evaluate or compile all of them. There are also commands to evaluate or compile an entire file.

Zmacs also knows about functions and variables in the Lisp environment. If you are in the middle of typing in a Lisp function call form, you can give a single-keystroke command to tell Zmacs to print out the argument list of the function. This is useful if you forget what arguments the function takes or in what order the arguments are expected. Another single-keystroke command prints out the on-line documentation associated with that function. If you have just typed in the name of a variable, another single-keystroke command asks Zmacs to tell you about that variable, giving its declaration and the documentation associated with it. These features provide instant confirmation of the correct spelling of names, as well as providing immediate access to documentation.

There is also a command that tells Zmacs to edit the definition of any Lisp function, variable, flavor, data structure, or other named entity. You can type the name of the thing to be edited on the keyboard, or you can point with the mouse at any name visible in the editor window. Zmacs will find the source text that defines that function, variable, or whatever, automatically reading in the file containing the text (if it isn't read in already), and position you at that definition so that you can examine and edit it.

The mouse can be used for many editing operations. You can point with the mouse to position the cursor somewhere in the text. You can mark out whole regions to be operated upon by sweeping over them with the mouse; with only mouse commands you can copy or move regions of text from one place to another. You can also use the mouse for scrolling; that is, to control which portion of the file is visible on the screen. You can control the speed of scrolling, and you can move to an arbitrary place in the file by graphically specifying how far into the file you want to see. You can also pop up a menu of some editor commands that are convenient to use the mouse for. Since Zmacs provides for positioning of the current cursor position with either keyboard commands or mouse commands, you can use whichever commands you personally prefer. For example, some people like to use the mouse for long-distance motion but prefer keyboard commands for short-range things like moving the cursor forward one or two characters or words.

Zmacs takes advantage of the bit-mapped display to provide faster and more convenient interaction with the user than is possible on a conventional terminal. The blinking of matching parentheses and the use of the mouse have already been mentioned. In the Emacs command set, many commands operate on a previously-selected region. In Zmacs, this region is underlined on the display, eliminating the danger of operating on the wrong region. Zmacs can edit text written in multiple fonts, with either fixed or proportional spacing.

Zmacs has a wide range of commands to offer the experienced user - over four hundred commands exist. While you only need a few commands to edit effectively, you can learn more and more commands, at your own learning rate, to let you do more powerful things. Here are a few examples of the more advanced commands:

- The Word Abbreviation facility lets you define short abbreviations for commonly-typed words or phrases; when you type in one of these abbreviations, it automatically and immediately expands into its full definition.

- In Auto-Fill Mode, Zmacs automatically inserts carriage return characters as you type at the ends of text lines, so that you can just type in one word after another and have them broken up into lines automatically.

- The List Functions command types out the names of all of the functions defined in the current file; you can then click on one of these with the mouse, and Zmacs will position you to the definition of that function.

- The List Combined Methods command understands the Flavors construct of Zetalisp; you give this command the name of a flavor, and the name of a message. It types out a list of all of the component methods that are combined to form the handler for the specified message to instances of the specified flavor. You may then click with the mouse on any of these names, and Zmacs will find the definition of that component method, read it in if it is not read in already, and position the cursor there so that you can edit it.

- The Electric Shift Lock Mode facilitates typing in programs that are in upper-case (if you like using upper-case in your programs). In Electric Shift Lock Mode, whenever you type a character that is part of a Lisp symbol, such as the name of a function, variable, or special form, it is inserted in upper case, but when you type a character that is part of a character string or a comment, it is inserted normally.

- The Zmacs sorting commands allow the text in the buffer or in a region to be sorted into alphabetical order, either line-by-line or with user-specified division into records.

You can customize your editing environment. Some commands have options and modes that specify details of their behavior, in cases where many users turn out to disagree about how they prefer the command to work; you can set each of these as you like it. You can also control the assignment of commands to keys, adding new commands and replacing other commands as you see fit, putting the commands you personally use most often on single keystrokes. You can also write your own editor commands. Since Zmacs is written completely in Lisp, you can use the same language that you use for your own programs to write editor enhancements, features, and commands. Zmacs is built on a large and powerful system of text manipulation functions and data structures, called *Zwei*. By writing your own Lisp programs that call Zwei functions to perform primitive text manipulation operations, you can write your own editor commands the same way that built-in commands are written. You can build your own library of commands, each with its own attached documentation, and then other users can load in this library, and assign your new commands to any keys they want to.

Any interactive program can provide the user with text editing capability simply by calling functions in the Zwei system to provide the full power of Zmacs. For example, the ZMail mail reading system uses Zwei functions to allow you to edit mail as you are sending it, or to edit mail that you have received. This sharing of large subsystems is unique to computer systems that provide a large, dynamically linked environment, as the Zetalisp environment does. Any application program can provide powerful editing to the user easily; the programmer doesn't have to write a new editor for each application. You only have to deal with one editor, instead of several, and any customizations that you make will take effect in all of the applications programs as well as in Zmacs itself.

There are several reasons that a character-oriented display editor like Zmacs is preferable to a list-structure editor such as is used in the Interlisp environment. The main reason is that a character-oriented editor gives you complete control over the *textual* appearance of your program. Sometimes the best indentation or textual layout of a piece of a program is dictated by something that the editor can't easily "understand"; for example, a Lisp list of elements might be representing something in a pairwise fashion, and so you might want it to appear textually as two columns of items. With Zmacs, you can indent a program any way you want, by simply not using the automatic indentation facilities when you don't want them. You can put in textual comments anywhere in the program, since they need not be part of the list structure of the program itself; this makes commenting easier and more convenient, and so high-quality commenting is encouraged. Your program appears exactly as you like it. Another advantage of the Zmacs approach is that you can use the same editor for editing text as you use for editing programs.

File Systems

The Symbolics system uses disk files as its basic long-term storage mechanism. Disk files need to be provided by a file system. There are two ways for the system to get at files. One way is to have a file system on the disk of your own machine. The other way is to access a computer with its own file system, called a file server, over the network; the file server can either be another Symbolics system, or an existing time-sharing system. Both kinds of file systems may be used alone or in conjunction with other file systems.

The machine's own file system can be used either as a local file system (a file system on your own machine), or in a remote file system (a machine that is connected to other machines over the network and provides a file system for them), or both. Directories are arranged in hierarchy up to sixteen levels deep. Names of files and directories may be as long as you like. Files have version numbers, and the generation retention count feature lets you tell the file system to automatically delete old versions as new ones are created. Undeletion is supported; files don't go away until a directory is "expunged". Directories may also contains "links", named entries which are pointers to other files elsewhere in the file system hierarchy. In addition to recording the usual attributes of files such as creation time, modification time, author, and length, the file system also keeps a property list with each file so that you can store your own attribute information with any file. You can add up to eight disks to a machine to create a very large file system.

The file system is very robust. All information in directories is fully redundant; if all of the directories in the file system were somehow destroyed, they could be completely re-created from the information stored in the "file headers" of the files. Every block of the file system is marked with a unique identifier so that the file system can check to make sure that when it has read a block, it has gotten the right data. All file system information is redundantly stored so that loss of a single block of the disk cannot damage more than one file. The file system is written using a transaction discipline so that you can continue using it after a crash without running a salvager. A salvager is provided for long-term recovery of "lost" blocks, rebuilding damaged directories, and otherwise automatically fixing file system problems. You don't need to have a file system expert around to fix things up after a crash. A magnetic tape backup system is also provided, supporting both incremental and complete dumps.

From a Symbolics system, you can access many other file systems over the network. These file systems can be provided by other machines used as remote file servers, or any of various other file systems provided by other computer systems. File systems currently supported are TOPS-20, Tenex, ITS, and VAX/VMS; any of these systems can be connected to a Symbolics system with the network. (Support for Unix on the PDP-11 and the VAX is currently being implemented.) Any command that prompts for a file name, or any function that takes a file name as an argument, can accept a file name for any file system, in that system's own syntax. For example, you can read files into editor buffers, or read them from any program, giving the file name in whatever syntax the file server uses. The file system hierarchy editor (described below) works on any file system, local or remote. Wildcard matching and automatic file-name completion are provided for all file systems, remote as well as local, even if these features are not present natively in the file system's time-sharing system.

Inspector, Display Debugger, and File System Editor

The Inspector is a tool for inspecting data structures. It displays a Lisp object showing all of its components. For example, if you inspect an array, the inspector displays the elements of the array; if you inspect a structure, it shows you all of the component slots of the structures, along with the names of the slots; if you inspect a symbol, it shows you the name of the symbol, the value, the property list, the associated function, and the symbol's package. If you inspect a list, it pretty-prints the list. The component objects are all mouse-sensitive. If you click on one of these components with the mouse, that component object gets inspected: it expands to fill the window and its components are shown. In this way, you can dive into a complex data structure, exploring the relationships between objects and the values of their components. The Inspector also keeps a history window recording the objects you have examined, so that you can back up and continue down another path. Objects being inspected can also be modified.

The Display Debugger provides a clear picture of the state of a Lisp process at the time of an error. It divides its area of the screen into several panes (sub-regions): a display of the stack history with a pointer to the selected stack frame; the names and values of the arguments to the selected stack frame; the names and values of the local variables of the selected stack frame; a command menu; a Lisp interaction window; an Inspector window and an Inspector history list. You can select a different stack frame by clicking on it with the mouse, and examine its arguments and local variables. The command menu includes commands to return an arbitrary value from any stack frame, to restart any function call in the stack, and to recover from the error and proceed if possible. The Display Debugger is interfaced to the Inspector so that you can inspect the various values you find in stack frames as well as the bodies of executing functions.

In the Symbolics system, the user manipulates the file system with a display-oriented tool called the file system editor. This editor displays a line with the name of every item in a directory. If one of these items is a subdirectory, you can "open" it by clicking on it with the mouse; the list of items will dynamically open up and the contents of the subdirectory will be displayed. You can then open its subdirectories, and so on. By opening and closing directories, you can poke around in the file system and see what is there. Once you have found a file, directory, or link on which you want to operate, you click on it and get a menu of useful operations for that kind of item. For a directory, you can delete or undelete it, expunge its contents, create new directories and links within it, rename it, invoke the dumper to save its contents on tape, and so on. For a file, you can delete or undelete it, look at its contents, rename it, dump it, and so on. When you open a directory you can optionally ask to see only those files that match a given wildcard specification, rather than all the files in the directory.

Electronic Mail: ZMail

In a modern computer science research environment, electronic mail is a *sine qua non*. The Symbolics system provides the most powerful electronic mail facility in existence: ZMail. ZMail is an interactive system for reading and sending mail. Its command interface is easy to learn and fast to use. You can start by learning a basic set of commands that give you all the power of conventional mail readers, with the streamlined user interface made possible by the use of the mouse and menus. ZMail provides many more advanced and powerful commands to help you keep track of and deal with large amounts of mail.

When you start reading your mail with ZMail, the top of the ZMail window contains a summary of the messages in your mail file, showing one line of descriptive text for each message. One message is the currently selected message, and its summary line is marked with an arrow. In the bottom of the ZMail window, you see the message itself. If either the summary or the message is too large to fit completely into its window, you can scroll the window by using the mouse. In between the summary and the message is a menu of commands, including commands to move to the next or previous message, delete or undelete a message, reply to a message, and send new mail. You can move through your messages with the Next and Previous commands, or by pointing at a line in the summary window and clicking the mouse. When you reply to mail, you can see the message to which you're replying in the top half of the window, and type in your reply in the bottom half. While you are typing in your mail, you are using the same text editor as you use to type in programs and text files, and so you have full editing capability.

You can read several mail files at a time, and move messages from one mail file to another; this lets you save mail about different topics into different mail files. You can select sets of messages on which some operation should be done; for example, you can select a set and then delete all the messages in the set, or move them all into some mail file. The set can be specified either by your explicitly saying which messages you want in the set (by pointing at their lines in the summary window), or on the basis of some criterion based on attributes of the message. The latter is accomplished using a "filter": a criterion for whether a message should or should not be accepted into a set. You can filter messages based on the sender, the recipients, the time of sending, the contents, whether you have replied to the message, or any of several other criteria. You can also combine these criteria using ANDs, ORs, and NOTs, to specify complex filters that can discriminate finely between messages.

ZMail is heavily customizable. Users can control the layout of the display, which recipients of a message receive copies of a reply to that message, the format of paper copies of mail files, and so forth. Users can define their own special purpose filters to access sets of messages according to their personal criteria. The contents of ZMail's menus and the defaults for commands with options may be customized. ZMail makes it easy to customize these things without having to understand anything about how they are implemented, by providing a display-based user-profile editor.

The Window System, as seen by the user

The bit-mapped display screen is managed by the *window system*. *Windows* are rectangular regions of the screen that may be fully visible, partially visible and partially covered, or wholly covered by other windows, like pieces of paper on a desk. You can use windows to control many tasks at once. Each task is represented by a different window, and you can switch between tasks by simply clicking on the task's window with the mouse. When a window is partially covered by another, you can click on the part of the window that is peeking out, and that window will come to the top and be fully visible. You can control the configuration of windows by using the Screen Editor, an interactive system that lets you create, destroy, move, and reshape windows.

Switching between tasks is very easy. Instead of having to give an "exit" command, return to a command processor, and give another command to switch tasks, you can just point and click. No information is lost when you switch between tasks, so you are free to switch from one thing to another whenever you want to without worrying about destroying valuable state.

This is particularly important when you are developing programs. For example, you can split the screen between an editor window in the top half of the screen and a Lisp interaction window in the bottom half. Then when you find a problem with your program, you just click on the editor window and start editing the correction. When you are finished editing, one simple editor command incrementally compiles those parts of the program that you have changed. Then you can click on the Lisp window and try the program again.

The window system is hierarchical: in the same way that the screen can be divided up into windows, a window itself can be further sub-divided into smaller windows. A window that is divided in this way is called a "frame", and the sub-windows are called "panes". When you move around or change the size of a frame, by using the Screen Editor for example, the panes correspondingly change their size and position as well. You can also use the Screen Editor to manipulate the panes within the frame.

You can put more than one bit-mapped display on the machine: the window system works equally well on all of them. All of the facilities provided by the window system, including menus and frames and the Screen Editor, work on any display, including color displays. The mouse can track on any screen.

You can always see documentation of what you can do with the mouse. Near the bottom of the main screen is a window with a line of text that tells you the present meaning of each of the mouse buttons. For example, when you move the mouse over an item of a menu, the mouse documentation line tells you what would happen if you were to click on it. As you move the mouse from one window to another, or as windows pop up or change configuration under the mouse, the mouse documentation line is changed to tell you the new meaning of the mouse buttons in the new context.

Other useful information is displayed at the bottom of the screen. Below the mouse documentation line is the *who-line*, which tells you the date and time, the user name of the user logged into the machine, the state of the current process (whether it is running, stopped, waiting for keyboard input, or waiting for any of various other things to happen), and the current package (in which Lisp expressions typed in from the keyboard will be read). The who line also tells you when the system is doing file transfers. You can see what file is being read or written, and how far the transfer has proceeded. If the console has been idle for more than five minutes, the who-line displays the idle time, so that if you come upon a machine with no user, you can see if it has been used recently or if it is sitting idle.

The Window System, as seen by the programmer

The window system has a clear and natural appearance to the programmer. Each window is a Lisp object, an *instance* of some *flavor* (see the description of the flavor system, below). To manipulate a window, you send it messages. Windows understand a wide variety of messages that do many different things. There are messages for examining and altering the shape and positions of windows. These come in simple varieties, in which you express the size in terms of either pixels or characters, as well as in more advanced varieties in which you can tell a window to move near some point or near the mouse, or next to some other window. There are messages to control the appearance of the borders and the labels of windows. Another set of messages controls the typing of text onto the window; you can control the cursor position, erase areas, type text in various fonts, insert and delete lines and characters, control the interline spacing, and so on. Windows also understand a large set of messages to perform primitive graphics operations such as drawing points, lines, filled-in rectangles, filled-in triangles, regular polygons, circles, curves described by a sequence of line segments, and cubic splines. Another important graphics primitive provided is BITBLT (sometimes known as RasterOp), which moves arbitrary rectangular sections of a picture between arrays and windows or within windows, combining bits using a logical operator. BITBLT, as well as line, rectangle, and triangle drawing, are implemented in microcode for high speed. Windows also support messages for input from the keyboard and the mouse, controlling the configuration of frames, which windows are visible and which are on top of which, and various other attributes.

In addition to these low-level functions for manipulating windows, the window system provides a set of high-level facilities that make it easy for a programmer to create an advanced, streamlined user interface to a program. By calling simple functions, you can create windows that let the user specify information in the manner that it easiest for the particular application. To let the user choose between one of several alternatives, you can create a menu. A menu can pop up and disappear as needed, or stay on the screen until the user disposes of it, or be one of the panes of a frame; several flavors of menus are provided for each of these needs. Another kind of menu provides for mode-setting, in the style of car-radio buttons; the menu stays around on the screen and the selected mode is highlighted with inverse-video. Another kind of menu lets you make many choices ("one from column A and one from column B").

There are other interfaces to let the user make choices. The Multiple Choice facility provides a window containing a bunch of items, one per text line. For each item, there can be several yes/no choices for the user to make. The window is arranged in columns, with headings at the top. The leftmost column contains the text naming each item. The remaining columns contain small boxes (called choice boxes). A "no" box has a blank center, while a "yes" box contains an "X". Pointing the mouse at a choice box and clicking the left button complements its yes/no state. The *Choose Variable Values* facility presents you with a set of Lisp variables and their values. You can change the values by pointing at them with the mouse. For a variable whose value is always one of a small set of things, Choose Variable Values displays all the possible choices with the currently-selected one in bold-face type; you can click on any choice to select it. For ordinary variables, you can type in the new value.

Other high-level facilities are provided. Many windows in the system respond to scrolling commands, given by the mouse, all working in a uniform and consistent fashion. This interface for scrolling is available to any program that wants it as a facility of the window system. This makes it easy for you to make your own windows handle scrolling, and encourages everyone to make all windows provide scrolling with the same uniform user interface. There are also facilities to let you easily specify regions of the window that should be mouse-sensitive; a little rectangular box lights up around such a region when the mouse moves into it, and when the mouse is clicked, the program is informed that said region has been clicked on.

Many interactive programs in the system construct their user interface by using the flavor system to combine (or "mix in") their own flavors with flavors chosen from the window system's extensive library of user interfaces. The private component flavors control or modify the behavior of the library flavors to meet the needs of the specific program. They may also take control over the way the window redisplays, how it responds to reshaping, and so on. A program may add its own messages and send them as well as the pre-defined window system messages.

Operating System

The Symbolics system does not have an "operating system" in the conventional sense, but all of the functionality that would normally be provided by an operating system is provided by some part of the software system. This is because the system is designed to be a single integrated Lisp environment for a single user, rather than a timesharing system that has to divide the computer between several users and protect each from the others. So the system provides operating system functionality with a structure designed to benefit the Lisp environment, rather than building a conventional operating system and then creating Lisp on top of that. The traditional boundary line between the operating system and the user program is deliberately blurred in the Symbolics system, making all predefined system facilities available to the user, making it easy for the user to modify or customize the system, and avoiding the need for one programming language and set of debugging tools for the system and another for the user.

Several processes can run concurrently, sharing the computational resources of the machine by using a scheduler. The scheduler is written in Lisp and implements processes using the *stack group* coroutining mechanism of the Zetalisp language. A process can wait for any arbitrary event to come true. To wait for an event, a process calls the scheduler's waiting function, passing it a Lisp function. The scheduler will periodically call this function, and as soon as it returns a true value, the process will be allowed to proceed. This is the fundamental waiting primitive; some higher-level facilities, such as sleeping for a given amount of real time and binary semaphores (locks), are provided as well. You can easily construct your own more powerful and complex multiprocessing control structures, by using Lisp macros. It is easy for processes to communicate data to one another, since they all exist in the same Lisp world; you can build any kind of mailbox or queueing or other facility that you need. (Some simple ones come pre-defined by the system.) The system itself uses processes heavily; one process reads keystrokes from the keyboard, processing interrupt characters and directing input to appropriate windows; two processes control the reception and transmission of packets on the network; the network remote login programs (Telnet and Supdup) work by splitting into two processes, one to transmit to the foreign host and one to receive from the foreign host. Users use processes as well; when you create a new window to run an interactive system, such as a Lisp Listener or a ZMacs (editor) window or a ZMail (mail reading) window, a new process is created to run that program. Lisp programs deal with processes by calling a function to create a process and then sending it messages to start, stop, and examine and alter the state of the process. There is also a simple Lisp function that calls a function on some arguments in its own process. Any time your program wants to do something that requires multiprocessing, it is easy for it to spawn any number of new processes to run concurrent programs. The scheduler is genuinely pre-emptive; it is co-called every second by a hardware-generated interrupt (the time-slice period is user-controllable).

The Symbolics system's virtual memory system is handled by the microcode; it is at a lower level than the Zetalisp language itself. A large, linear virtual memory is provided, managed by a paging algorithm that simulates the Least-Recently-Used page replacement principle using the standard clock algorithm. The virtual memory is split up into areas by the area feature of Zetalisp, and further divided into Lisp objects by the basic storage discipline on which Zetalisp is built.

The Chaosnet network control program provides Zetalisp programs with full access to the facilities of the network. Lisp functions are provided to open network connections as a user or to handle incoming network connections as a server. You can accept, reject, and close connections, and transmit or receive packets. You can also deal with a network connection as an I/O stream, and perform standard input and output operations on one. You can invent your own higher-level network protocols or use existing ones. The network control program provides error-checking, flow-control, and retransmission, and maintains the multiplexing and demultiplexing of the network into many separate connections. The Chaosnet also has provisions for simple transactions (a single exchange of packets, with acknowledgement, without forming a connection), uncontrolled packets (without flow control and acknowledgement), broadcasting, automatic redistribution of routing information for gateways, and error reporting. Higher-level protocols exist for remote login, file access, mail transmission, interactive messages, inquiring what users are using a machine, finding out the current time, accesing remote printers, gating to remote networks, and network maintenance and testing. The Chaosnet can also be used for the transmission of packets in foreign protocols (such as DOD Internet). Network control programs also exist for the Tenex, TOPS-20, ITS, VAX/VMS, and Unix (TM Bell) (both PDP-11 and VAX) operating systems. Support for the Ethernet II (TM Xerox) will be provided.

Miscellaneous Features

Other languages

In addition to the native ZetaLisp language, Symbolics will offer support for Fortran-77 and Pascal, allowing users to use packages written in these languages. A compatibility package for InterLisp users to import their programs will be supplied. More complete support for InterLisp and compilers for other popular languages (C, Ada, etc.) are being considered for future development.

Remote login

You can log into other computers over the network. The Symbolics system supports the standard Telnet remote-login protocol as well as the Supdup display-terminal oriented remote-login protocol. If you are an Arpanet user, the system can connect to hosts over the Arpanet, using a gateway.

Interactive messages

You can send interactive messages to other users, over the network. When the system receives a message, it pops up a window, displays the message in the window, and prompts you with "Reply?". If you type "N", the window just disappears; if you type "Y", you can then type a message and it will be sent off to the original sender.

Examining the system

The Peek utility program provides a window-oriented interface that lets you see what is going on inside the system. You can look at a listing of all the processes, seeing what state each one is in; you can see the window system hierarchy; you can see how memory is being utilized in the different areas; you can see all the open network connections; you can get the status of other sites on the network; and you can look at general system meters. All of these displays update themselves continually, so you can watch as processes change state, windows get created, destroyed, and reorganized, and so on. You can also click on these displays to perform simple operations on the objects; for example, you can start and stop the processes, and you can rearrange and destroy the windows.

Metering

The system includes performance-analysis tools which can be used to optimize the performance of user programs, in terms of both processor utilization and virtual memory paging.

The Zetalisp Language

Zetalisp is a modern, powerful dialect of Lisp. It is based on the M.I.T. Maclisp dialect, and has been substantially extended and improved. Over the course of many years, Zetalisp has integrated features of Interlisp, other artificial intelligence languages, and a variety of modern, conventional systems programming languages, as well as facilities developed by its users. The resulting language is the most powerful base available today for both artificial intelligence research and systems programming in general. The following sections outline some of the important features of Zetalisp.

A Rich Environment

The most important advantage of Zetalisp is that it provides an interactive, incremental, dynamically linked programming environment. When you set out to write a program in Zetalisp, you don't start in a vacuum, with only a fixed set of simple language primitives to build on. Instead, your program is an addition to a rich environment full of useful facilities that you can invoke with a simple Lisp function call. If you want to sort a list or an array, you just call the "sort" function, which provides a carefully-optimized quicksort; if you want a hash table for something, you just call the "make-hash-table" function, and then use simple Lisp functions to insert elements and look them up. You can get pseudo-random numbers by calling "random". You can convert dates and times between various formats, and print them out or read them in, by calling a set of already-existing functions. You can can solve systems of linear equations and do matrix arithmetic by calling the functions provided. All these things are at your fingertips whenever you write a program; you don't have to write your own sorting, hashing, or random number generators every time.

Data Types

Zetalisp supports a wide variety of data types, and provides for user-defined data types as well. Basic type checking is done in microcode for all operations, so that taking car of a number or trying to add a symbol will never go undetected, even in compiled code; type mismatches always signal an error.

Arrays are a fully-supported data type. Arrays can be created, stored in data structure, and passed around as arugments. Simple Lisp functions make arrays and examine and alter their elements. Special types of arrays exist to store fixed-point numbers in a packed format. Arrays can have up to seven dimensions.

Character strings are also supported as a first-class data type. A character string is a kind of array, so the regular array accessing functions can be used to manipulate the characters of a string directly. In addition, an extensive set of string manipulation functions is supported. Functions are provided to search a string for a character, a substring, or a set of characters, forward or backward; to concatenate, take substrings, reverse strings, trim specified characters off the ends, and so on. You can use Zetalisp's stream-directed input/output to print to strings or read from strings.

Fixed-point, floating-point, and arbitrary-precision fixed-point ("bignum") numbers are supported. All standard arithmetic functions are generic, working correctly with any numeric data type and performing type coercions at runtime when necessary.

Two forms of user-defined data type are provided. The first kind is simply a structure (record) with named components, each of which may be any Lisp object. The programmer can specify the type identifier for such structures, and control their printed representation so that they can print out in a descriptive way. The second kind is more powerful; it is the "flavor" system, described in detail below.

Control Structures

Zetalisp has all of the control structures employed by conventional languages. It has simple conditionals, multi-armed conditionals, and several kinds of dispatching ("case") constructs. There are several iteration constructs: "dolist" and "dotimes" for simple iteration over a list or a sequence of numbers, the Maclisp "do" for more general iteration allowing stepping of many variables in parallel and arbitrary end-test, and the extremely general and powerful keyword oriented "loop" construct. The "prog" feature with gotos is also supported, although it is rarely needed.

There are also more advanced control structures. One of these is the "catch/throw" mechanism for structured non-local exits. The body of a catch form is evaluated and returned, except that if, during that evaluation, a throw with a tag matching that of the catch gets evaluated, then the catch immediately returns the value given to the throw function. One of the primary uses of this mechanism is to quit out of a program when an error is detected. It can also be used, for example, to get out of levels of looping and recursion upon finding an item for which you are searching. Zetalisp also provides the "unwind-protect" special form, which evaluates Lisp forms in such a context that if a non-local exit is done from those forms, a "clean-up handler" will be evaluated. By setting up unwind-protect forms around appropriate places, you can arrange for temporary effects to be undone whenever an evaluation finishes, whether it finishes normally or by means of a non-local exit. This means that you can arrange for things to be cleaned up if someone aborts a process or the process gets an error that is dismissed.

Zetalisp also provides two kinds of coroutining control structures. Simple coroutines can be created using "closures". A closure is a Lisp function along with some saved variable-binding state information. A function can be written as a generator that saves its state in some variables, and then by creating a closure of that function over those variables, you can have a new instance of the generator. More powerful and general coroutines can be created using "stack groups". A stack group holds the entire state of an executing Lisp program including the control stack history and the state of the bound variables. At any time, one stack group is the currently executing stack group. It can co-call another stack group, which transmits a value to that stack group and starts its computation running. Multitasking, implemented using stack groups and a scheduler, is also provided, and was discussed above.

Function Calling

Zetalisp provides more powerful argument passing facilities for functions than ordinary Lisp dialects. A function may have any number of required parameters, followed by any number of optional parameters, followed optionally by a "rest" parameter. Whenever a function call is done, the number of arguments is always checked for validity, and if the caller passes too few or too many arguments for the callee, an error is signalled. Optional arguments may be optionally supplied by the caller; the callee can specify an initialization form that provides the value of the optional parameter if it is not passed. If the caller passes more arguments than all of the required and optional parameters, and the callee specifies a "rest" parameter, then the "rest" parameter's value will be a list of all of the rest of the arguments. This lets you write functions that take an arbitrary number of parameters.

Zetalisp lets you return more than one value from a function. This is useful when a function computes more than one interesting value. In ordinary Lisp dialects, you would have to create a data structure, such as a list, containing the values, and return it; the caller would then have to take this apart again. In Zetalisp, any function can return more than one value, and the caller can call the function requesting many values and specifying variables that should be bound or set to the values.

Input/Output

Input and output in Zetalisp are done through streams. A stream is a Lisp object that can act as a source and/or sink of sequential characters. An input stream object is sent messages such as "read the next character", "read a line", "are characters available?", "clear buffered input", and so on; an output stream object is sent messages such as "write this character", "write this string", "clear buffered output", "wait for buffered output to finish going to device", and so on. Streams are provided for communication with the keyboard, windows on the screen, files in any accessible file system, network connections, editor buffers, and character strings. All input functions work on any input stream, and all output functions work on any output stream (streams may be unidirectional or bidirectional), and you can hook any of them together in arbitrary patterns. You can easily write your own streams that deal with characters stored anywhere you want.

Zetalisp provides a function called "format" for doing formatted output conveniently; it is similar to the "printf" function in Unix. Format takes a string of text to be printed. The string can contain escape sequences, that generally say "take the next Lisp object that was an argument to format, and print it this way". You can print numbers in any base, with specified field widths and padding characters; you can also print English cardinal and ordinal numbers, and Roman numerals. Format also has provisions for pluralizing words, spacing to particular columns, outputting control sequences, conditionalizing text, iterating over lists, arbitrarily justifying fields, and so on. The output produced by format can be sent to a character string or to any output stream.

Managing Large Systems

In recent years, it has been recognized that the major problem in software is that programs are very complex, and that the job of a program development system is to help the programmer reduce and manage that complexity. Zetalisp provides a battery of language features and interactive tools to make programs easier to work with. This section describes some of them.

Large Lisp programs are usually divided up into several different files, to provide managable-sized pieces for text editing and compiling to files. Such a set of files can be presented to Zetalisp as a "system" and managed as a unit. A system declaration lists all of the files in a system and specifies their interdependencies on one another. In Lisp programs, for example, one file might provide macros that must be loaded into the Lisp environment in order for a second file to be compiled correctly. You can ask for a system to be recompiled, and the system will automatically find all of the files that have been modified since the last time they were compiled, and offer to compile them, first performing any actions that are required by the dependencies, such as loading files of macros. Recompilation can be done selectively, with the user queried for each file to be compiled. You are asked whether the file should be compiled, and presented with a directory listing showing the existing versions of the file with their creation times and authors, and you are offered a source-to-source incremental comparison between the installed version and the latest version, or between any versions you want, so that you can examine what changes have been made to the software and audit these changes before approving the recompilation.

Often bugs are found in large systems, and it is desirable to be able to fix the problem and distribute the correction to any users of the system. If the system is a small program that is loaded into the Lisp environment by each user every time it is used, there's no problem; you just recompile the files and users will get the lastest version when they load them. However, if the system is large and versions of it are stored away in saved Lisp environments, it is time-consuming to completely reload the system. Zetalisp provides a facility whereby the maintainers of the system can create "patch" files containing Lisp forms that fix the problem, and make these patches publicly available. Then a user can give a command that automatically loads up all of the lastest patches to the system into his environment. The patch system automatically keeps track of numbering patches, and producing two-part version numbers of the system based on the original version of the program that was loaded and the level of patches that have been installed. Simple commands in Zmacs ("Add Patch" and "Finish Patch") make it easy to install new patches to a system.

The Zetalisp environment is based on a single, large "workspace" containing all of the functions and global variables of all of the programs in the system. In order to prevent naming conflicts between different programs that happen to both choose the same name for some function or variable, the system provides for independent namespaces for different programs. A namespace is an association between character strings and Lisp symbols; two namespaces can associate the same character string with two different symbols. Any program can have its own namespace so that it can choose the names it wants to use without getting in the way of any other program.

There are several interactive facilities to help the programmer examine and debug programs. You can trace all function calls to a particular function or set of functions, optionally causing breakpoints. You can find out all of the callers of a particular function, in case you want to change the way that function behaves and you need to change all the callers. You can find all of the symbols whose name contains a given substring; this is useful if you only remember part of a name. You can set up a region of memory so that if there is any attempt to write it, the error handler will be called; this is useful if something somewhere is incorrectly writing something and you want to find the culprit. You can single-step a program, watching each evaluation as it happens and examining the environment between steps. You can find out the argument list of any function loaded into the environment, and find the attached on-line documentation for the function if the author wrote any.

Flavors: Message-passing.

Object-oriented programming is available in Zetalisp. In this style, a program is built out of a number of objects. Operations are done on objects by sending them messages. Each object understands a certain set of messages, and responds to those messages in a certain way. Message passing is powerful because different objects can respond to the same set of messages in different ways. For example, input/output streams in Zetalisp use message passing. Each stream is represented by a Lisp object that accepts the stream messages, such as "input the next character" and "output this character string". A program that performs input/output is given a stream, and it sends these messages to the stream. Different streams handle the messages in different ways; for example, one stream might handle "output this character string" by displaying it on a window on the screen, another might send the string out to the network, and a third might insert the string into an editor buffer. The program that did the output need not know what the stream will do; it just sends the messages.

Sending a message from a Zetalisp program is easy; you just invoke the receiving object as a function. The first argument to the function is the name of the message, and the rest of the arguments are arguments to the message. Any Lisp object that accepts these arguments can be a receiver of messages. Zetalisp provides a facility called Flavors that makes it easy to create Lisp objects that receive messages. A flavor is a kind of user-defined data type. When you define a flavor, you have defined a new data type; you can create new objects of that type, which are called *instances*. For each flavor, there is a set of *instance variables*; each instance has its own set of values for these variables. Instance variables hold the internal state of each instance. Each flavor also has a set of Lisp functions called *methods*, of which there is one for each message that instances of this flavor understand. When a messages is sent to an instance, the Lisp system finds out what flavor the instance is an instance of, and finds the method of that flavor for the message being passed. This method is then invoked, and it is passed the arguments of the original message. The method can be any Lisp function, and it computes and returns a result to the sender of the message.

The great power of flavors is that you can combine several flavors to form a new flavor. You can take a basic flavor that implements some new data type, and build a new flavor, adding new functionality and modifying old behavior, by mixing the basic flavor with other flavors. For example, windows in the window system are represented as instances of flavors. There are many different flavors of windows. One flavor might be a basic window with other flavors "mixed in" to provide for such features as borders, labels of various sorts, ability to do graphics output, and so on. Flavors can be mixed in any fashion you want. When two flavors provide methods for the same messages and a new flavor is built on them, the two methods are combined, in any of several ways; one may override the other, they may both be performed in some order, one may be executed only if the other says that it should be, and so on, under complete control of the programmer.

Macros: Extending the Language

In most languages, there is a fixed set of syntactic constructs that are part of the language; you get them with the language and those are all you get. In Zetalisp, you can add your own constructs to suit your particular needs or tastes, or to tune the language to a particular application. You do this by creating a Lisp *macro*. Macros in Zetalisp are different from macros in the traditional sense; rather than manipulating text, they manipulate the structure of the program. A Lisp program is made of Lisp data structure, with lists, symbols, numbers, and so on; a Lisp macro is a Lisp function that manipulates the structure of a Lisp program, translating the syntax that you invent for your extension into existing Lisp constructs. When you build a large software system in Zetalisp, you usually create several language extensions that are specific features useful for your system, and then you write your system in this extended language. Specialized languages have been built on Lisp for such diverse purposes as computer graphics, text formatting, expert problem solving, and VLSI integrated circuit design. Zetalisp includes several tools and constructs to make it easy and convenient to write macros.

The Zetalisp Dialect

Zetalisp is a new Lisp dialect, based on the best features of Maclisp and Interlisp, and modified by years of experience in writing and maintaining large programs in Lisp. Zetalisp is largely upward-compatible with Maclisp; most Maclisp programs can easily be made to run in Zetalisp as well as Maclisp. Zetalisp will be a superset of the Common Lisp standard Lisp subset, which will allow programs to be portable to several other modern Lisp systems.

Zetalisp includes many features that are familiar to Interlisp programmers but which have not traditionally been provided in Maclisp. For example, functions in Zetalisp can be customized and modified by the addition of user-specified code called "advice". Structures (records) with named slots can be defined and used. A powerful iteration construct using keywords to express various kinds of looping and actions to be taken is provided. Many other little things like hash tables have also been added to Zetalisp.

Zetalisp was designed from the beginning with Maclisp compatibility as a goal, to allow researchers with existing Maclisp software to run their programs in the Zetalisp environment. Several large systems currently run in both Maclisp and Zetalisp, with only a small amount of conditionalized code.

A common subset of Lisp, called *Common Lisp*, is being defined by the designers of several new Lisp dialects, to provide portability between the different dialects. The SPICE Lisp dialect, being developed as part of the Carnegie-Mellon University SPICE project, and the implementations of NIL (New Implementation of Lisp) for both the Lawrence Livermore National Laboratories S-1 and the DEC VAX, will also include the Common Lisp subset. Any program written wholly in this subset will run correctly in any of these dialects.

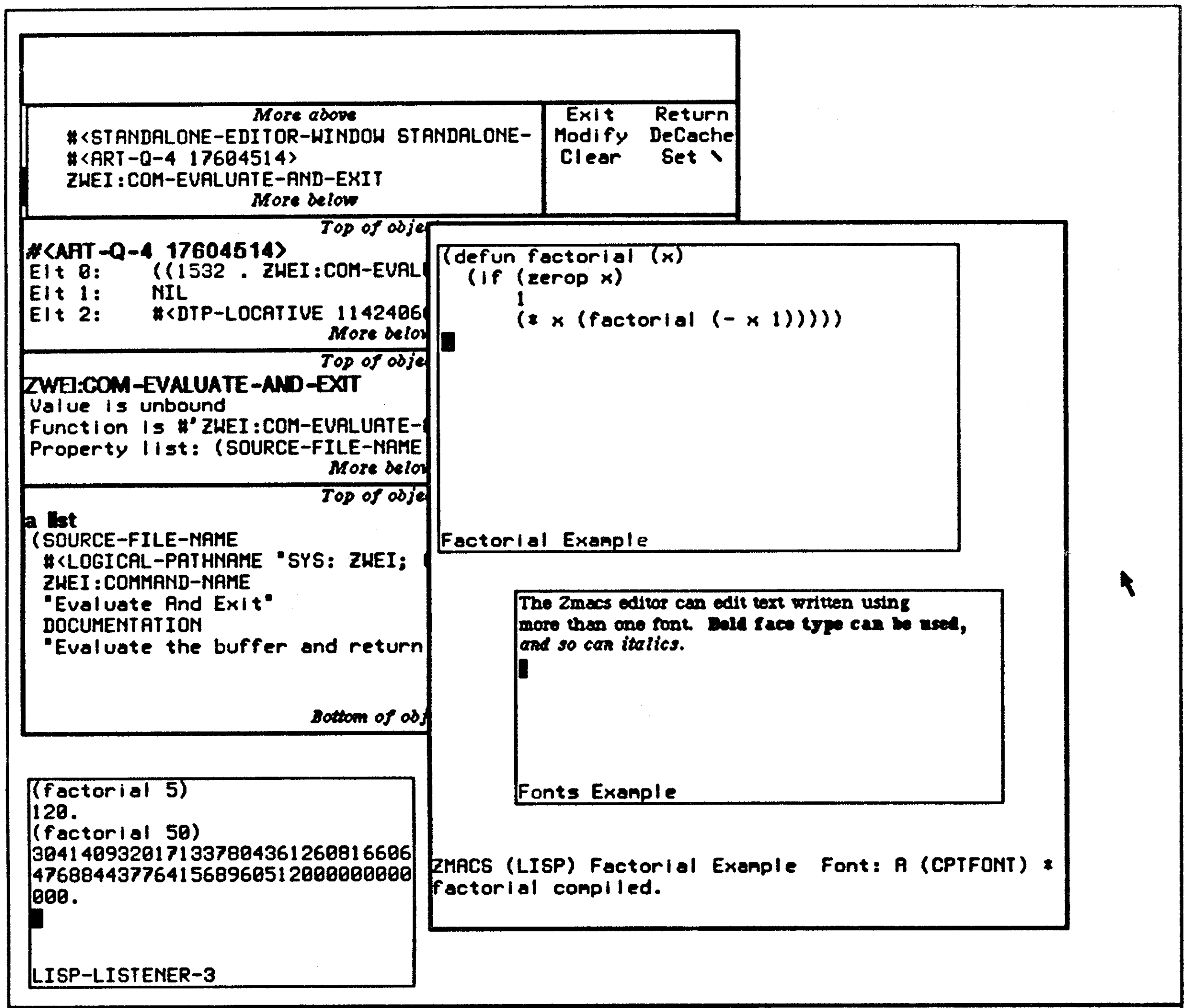
Common Lisp will be well-supported by all of the above-mentioned dialects; the participants are all committed to providing and maintaining support for the entire subset. Common Lisp will also be very stable; additions to the definition will only be made if there is general agreement among all participants that such an addition should be made, and incompatible changes will be avoided. New Lisp language features that are still considered to be under development will not be added to Common Lisp; only stable and well-tested features will be added.

Other Language Features

Zetalisp gives you access to the lowest levels of the software system and the processor, if you need them. A special set of functions called *subprimitives* can be used to manipulate internal data structures and deal with levels lower than the Zetalisp language itself. Many of the fundamental functions and facilities of Zetalisp are written as Lisp programs that call these subprimitives. While it is rare that user programs need to use these facilities, they are present if you ever need them.

Errors detected either in microcode or by Lisp programs are reported to the user with clear, English error messages. Whenever an error is signalled and not handled by the program, an interactive error handler gives you the error message and provides you with commands to examine and manipulate the state of the program. This error handler has a command to invoke the Display Debugger.

Programs can set up handlers for exceptional conditions. Conditions are signalled by Lisp errors, or by any program that wants to signal one. Errors can be dealt with and program execution resumed, or the program can quit what it is doing and take other actions based on the error. A handler can be set up to catch a specific error, any of a class of errors, or any error at all. Any program can create its own errors and classes of errors, and they will be handled uniformly by the system. (The condition handler system is not yet fully implemented.)



```
More above
  #<STANDALONE-EDITOR-WINDOW STANDALONE-
  #<ART-Q-4 17604514>
  ZWEI:COM-EVALUATE-AND-EXIT
More below
```

```
Top of obje
#<ART-Q-4 17604514>
Elt 0:     ((1532 . ZWEI:COM-EVAL
Elt 1:     NIL
Elt 2:     #<DTP-LOCATIVE 1142406
More belo
```

```
Top of obje
ZWEI:COM-EVALUATE-AND-EXIT
Value is unbound
Function is #'ZWEI:COM-EVALUATE-
Property list: (SOURCE-FILE-NAME
More belo
```

```
Top of obje
a list
(SOURCE-FILE-NAME
 #<LOGICAL-PATHNAME "SYS: ZWEI;
 ZWEI:COMMAND-NAME
 "Evaluate And Exit"
 DOCUMENTATION
 "Evaluate the buffer and return

Bottom of obj
```

```
(defun factorial (x)
  (if (zerop x)
      1
      (* x (factorial (- x 1)))))

Factorial Example
```

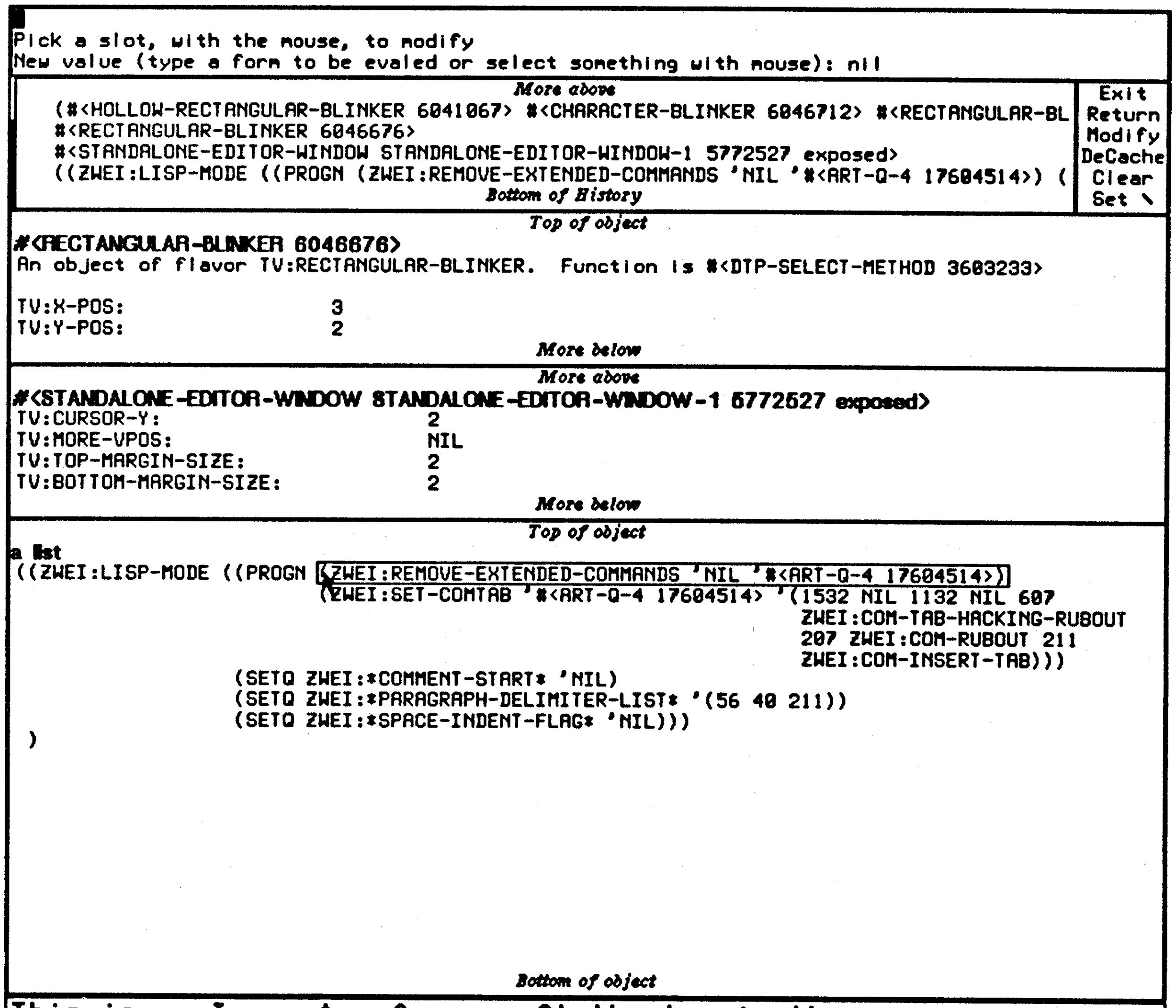
The Zmacs editor can edit text written using
more than one font. **Bold face type can be used,**
and so can italics.

```
Fonts Example
```

```
ZMACS (LISP) Factorial Example  Font: A (CPTFONT) *
factorial compiled.
```

```
(factorial 5)
120.
(factorial 50)
3041409320171337804361260816606
4768844377641568960512000000000
000.

LISP-LISTENER-3
```

The Zmacs frame has two windows. In one, a factorial function has just been written and compiled; the other demonstrates the use of fonts. There is also a Lisp Listener window in which the factorial function is tried out, and an Inspector frame.



```
Pick a slot, with the mouse, to modify
New value (type a form to be evaled or select something with mouse): nil
```

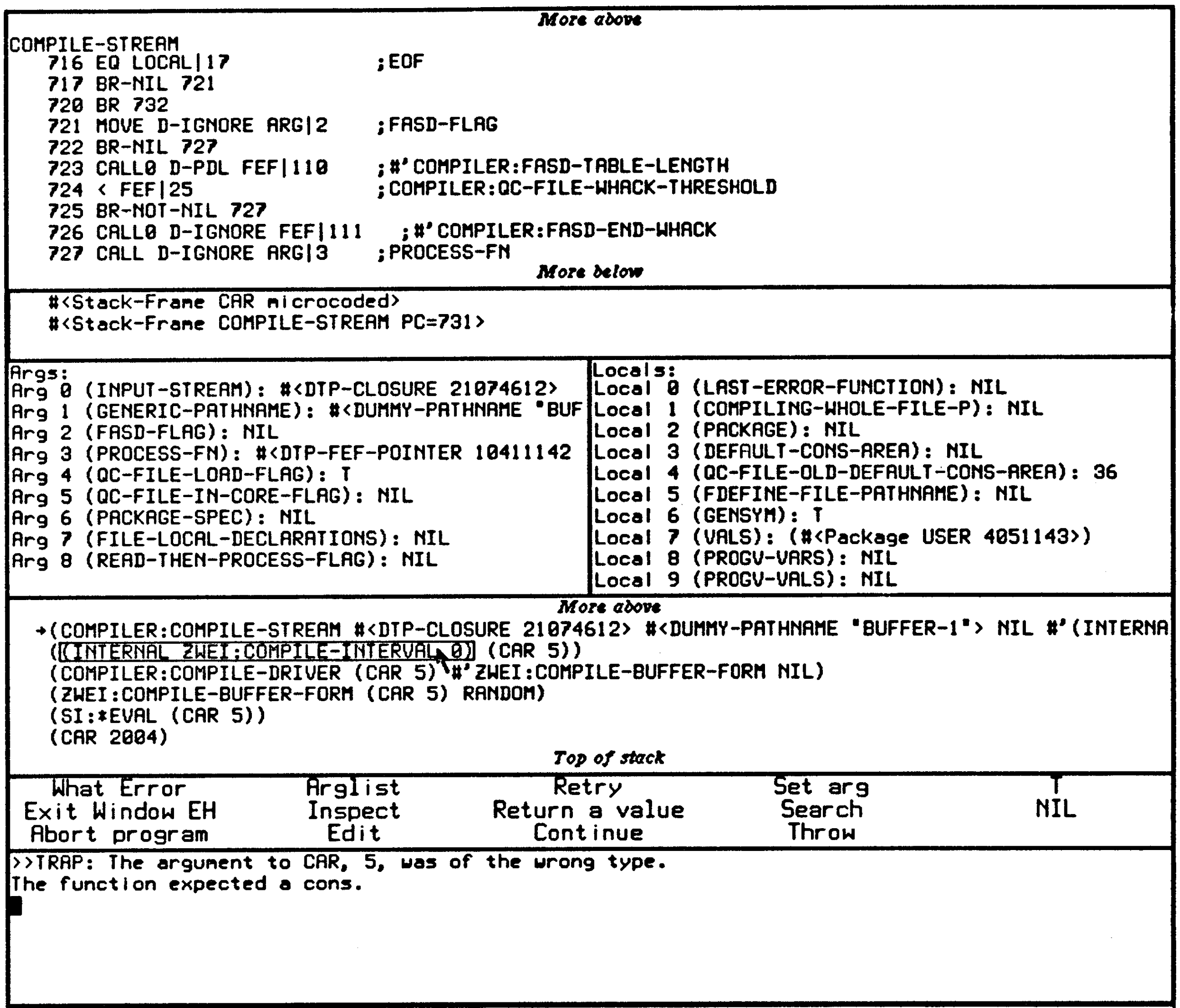
```
(#<HOLLOW-RECTANGULAR-BLINKER 6041067> #<CHARACTER-BLINKER 6046712> #<RECTANGULAR-BL
#<RECTANGULAR-BLINKER 6046676>
#<STANDALONE-EDITOR-WINDOW STANDALONE-EDITOR-WINDOW-1 5772527 exposed>
((ZWEI:LISP-MODE ((PROGN (ZWEI:REMOVE-EXTENDED-COMMANDS 'NIL '#<ART-Q-4 17604514>) (
```

```
#<RECTANGULAR-BLINKER 6046676>
An object of flavor TV:RECTANGULAR-BLINKER.  Function is #<DTP-SELECT-METHOD 3603233>

TV:X-POS:              3
TV:Y-POS:              2
```

```
#<STANDALONE-EDITOR-WINDOW STANDALONE-EDITOR-WINDOW-1 5772527 exposed>
TV:CURSOR-Y:                  2
TV:MORE-VPOS:                 NIL
TV:TOP-MARGIN-SIZE:           2
TV:BOTTOM-MARGIN-SIZE:        2
```

```
a list
((ZWEI:LISP-MODE ((PROGN (ZWEI:REMOVE-EXTENDED-COMMANDS 'NIL '#<ART-Q-4 17604514>)
                         (ZWEI:SET-COMTAB '#<ART-Q-4 17604514> '(1532 NIL 1132 NIL 607
                                                                  ZWEI:COM-TAB-HACKING-RUBOUT
                                                                  207 ZWEI:COM-RUBOUT 211
                                                                  ZWEI:COM-INSERT-TAB)))
                  (SETQ ZWEI:*COMMENT-START* 'NIL)
                  (SETQ ZWEI:*PARAGRAPH-DELIMITER-LIST* '(56 40 211))
                  (SETQ ZWEI:*SPACE-INDENT-FLAG* 'NIL)))
 )
```

This is an Inspector frame. At the top is the interaction pane; below is a history pane and a command menu. Below them are three Inspect panes, each inspecting one Lisp object. The first object is a blinker, the second is a window, and the third is a list.



```
COMPILE-STREAM
   716 EQ LOCAL|17          ;EOF
   717 BR-NIL 721
   720 BR 732
   721 MOVE D-IGNORE ARG|2  ;FASD-FLAG
   722 BR-NIL 727
   723 CALL0 D-PDL FEF|110  ;#'COMPILER:FASD-TABLE-LENGTH
   724 < FEF|25             ;COMPILER:QC-FILE-WHACK-THRESHOLD
   725 BR-NOT-NIL 727
   726 CALL0 D-IGNORE FEF|111  ;#'COMPILER:FASD-END-WHACK
   727 CALL D-IGNORE ARG|3  ;PROCESS-FN
```

```
   #<Stack-Frame CAR microcoded>
   #<Stack-Frame COMPILE-STREAM PC=731>
```

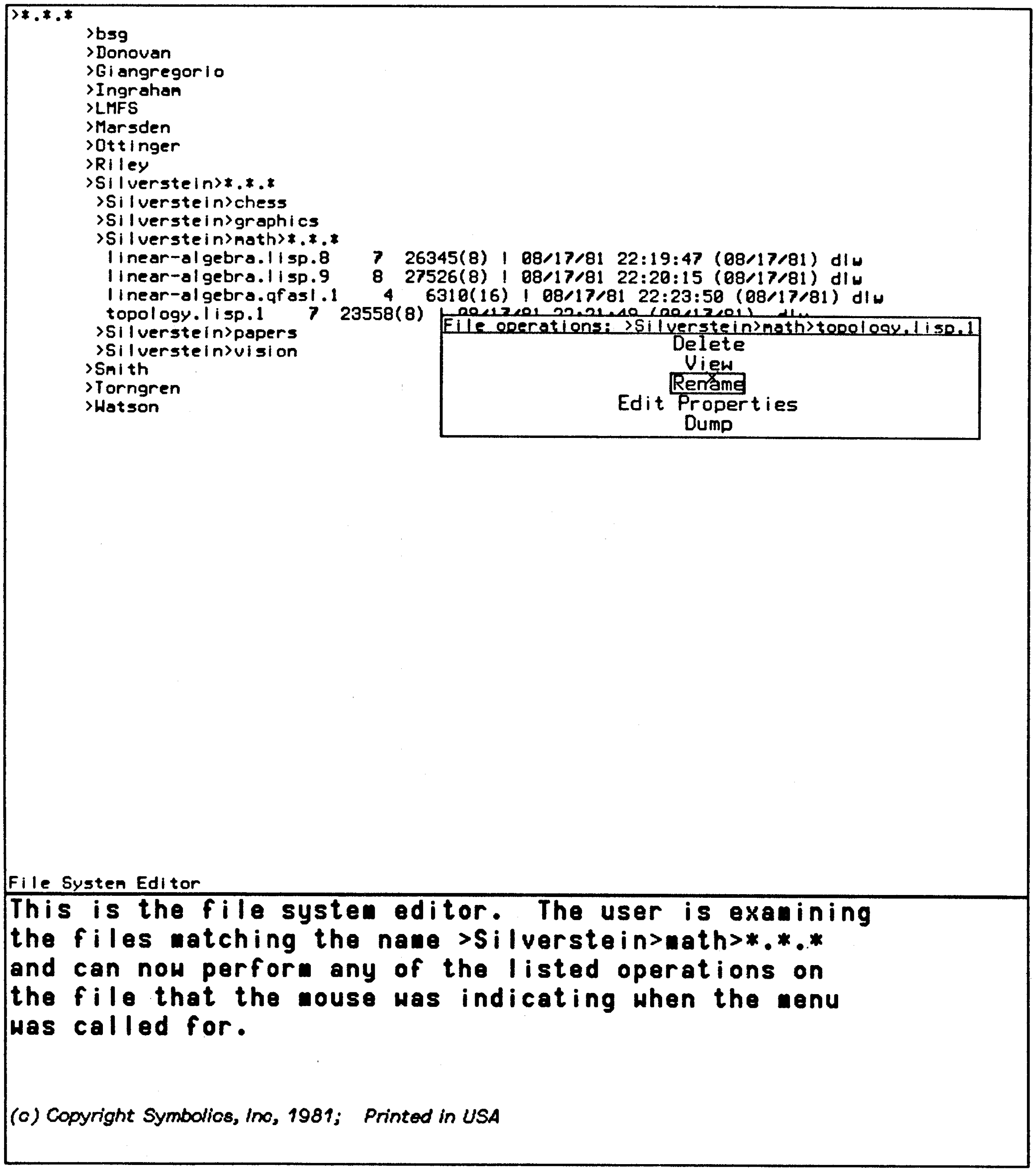
```
Args:
Arg 0 (INPUT-STREAM): #<DTP-CLOSURE 21074612>
Arg 1 (GENERIC-PATHNAME): #<DUMMY-PATHNAME "BUF
Arg 2 (FASD-FLAG): NIL
Arg 3 (PROCESS-FN): #<DTP-FEF-POINTER 10411142
Arg 4 (QC-FILE-LOAD-FLAG): T
Arg 5 (QC-FILE-IN-CORE-FLAG): NIL
Arg 6 (PACKAGE-SPEC): NIL
Arg 7 (FILE-LOCAL-DECLARATIONS): NIL
Arg 8 (READ-THEN-PROCESS-FLAG): NIL
```

```
Locals:
Local 0 (LAST-ERROR-FUNCTION): NIL
Local 1 (COMPILING-WHOLE-FILE-P): NIL
Local 2 (PACKAGE): NIL
Local 3 (DEFAULT-CONS-AREA): NIL
Local 4 (QC-FILE-OLD-DEFAULT-CONS-AREA): 36
Local 5 (FDEFINE-FILE-PATHNAME): NIL
Local 6 (GENSYM): T
Local 7 (VALS): (#<Package USER 4051143>)
Local 8 (PROGV-VARS): NIL
Local 9 (PROGV-VALS): NIL
```

```
  →(COMPILER:COMPILE-STREAM #<DTP-CLOSURE 21074612> #<DUMMY-PATHNAME "BUFFER-1"> NIL #'(INTERNA
   ((INTERNAL ZWEI:COMPILE-INTERVAL 0) (CAR 5))
   (COMPILER:COMPILE-DRIVER (CAR 5) #'ZWEI:COMPILE-BUFFER-FORM NIL)
   (ZWEI:COMPILE-BUFFER-FORM (CAR 5) RANDOM)
   (SI:*EVAL (CAR 5))
   (CAR 2004)
```

```
>>TRAP: The argument to CAR, 5, was of the wrong type.
The function expected a cons.
```

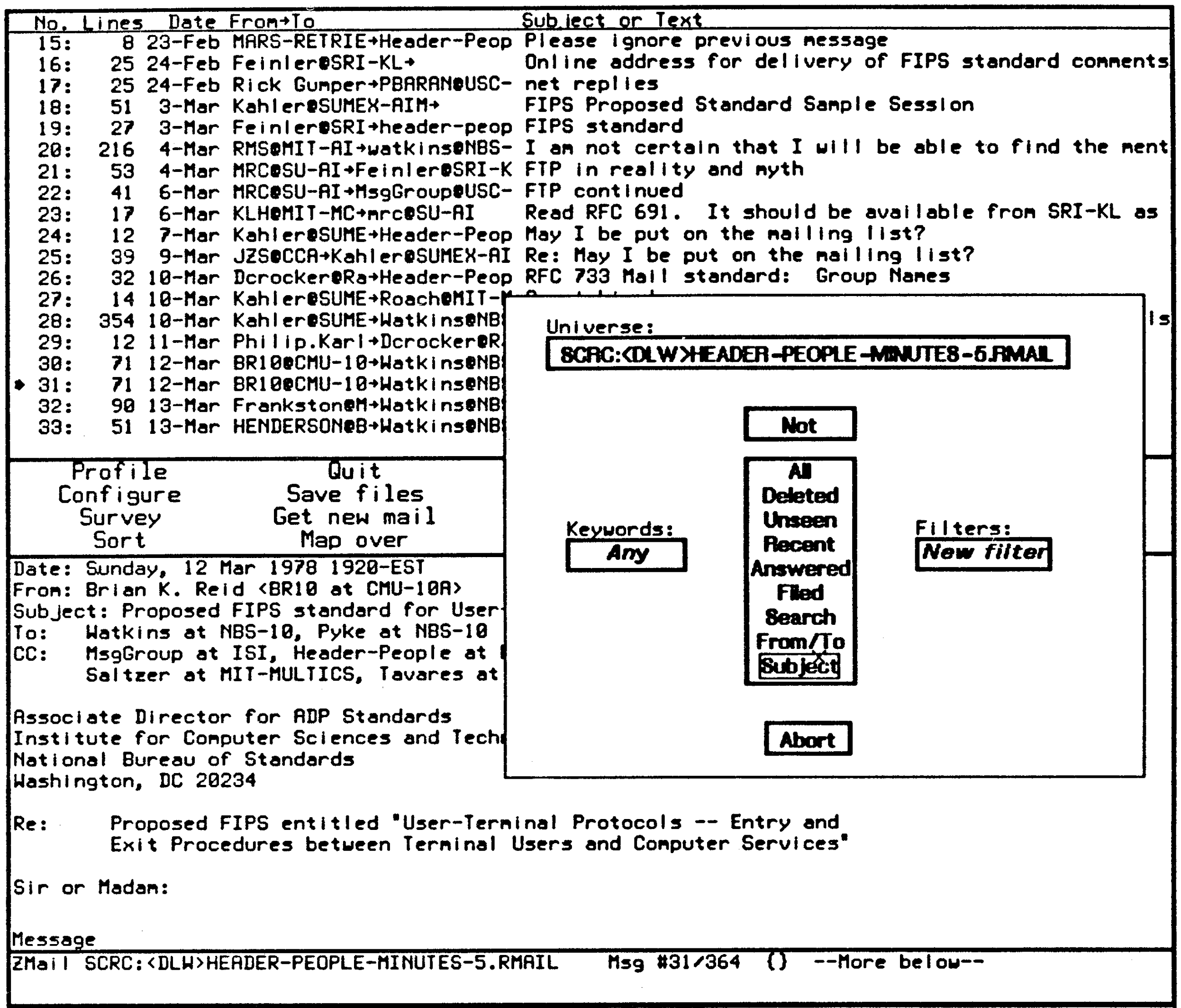
This is the Display Debugger. It shows a stack history, including the arguments, local variables, and code for the currently selected stack frame. There is also a command menu and an interaction pane.



```
>*.*.*
        >bsg
        >Donovan
        >Giangregorio
        >Ingraham
        >LMFS
        >Marsden
        >Ottinger
        >Riley
        >Silverstein>*.*.*
         >Silverstein>chess
         >Silverstein>graphics
         >Silverstein>math>*.*.*
          linear-algebra.lisp.8    7  26345(8) ! 08/17/81 22:19:47 (08/17/81) dlw
          linear-algebra.lisp.9    8  27526(8) ! 08/17/81 22:20:15 (08/17/81) dlw
          linear-algebra.qfasl.1    4   6310(16) ! 08/17/81 22:23:50 (08/17/81) dlw
          topology.lisp.1    7  23558(8)
         >Silverstein>papers
         >Silverstein>vision
        >Smith
        >Torngren
        >Watson
```

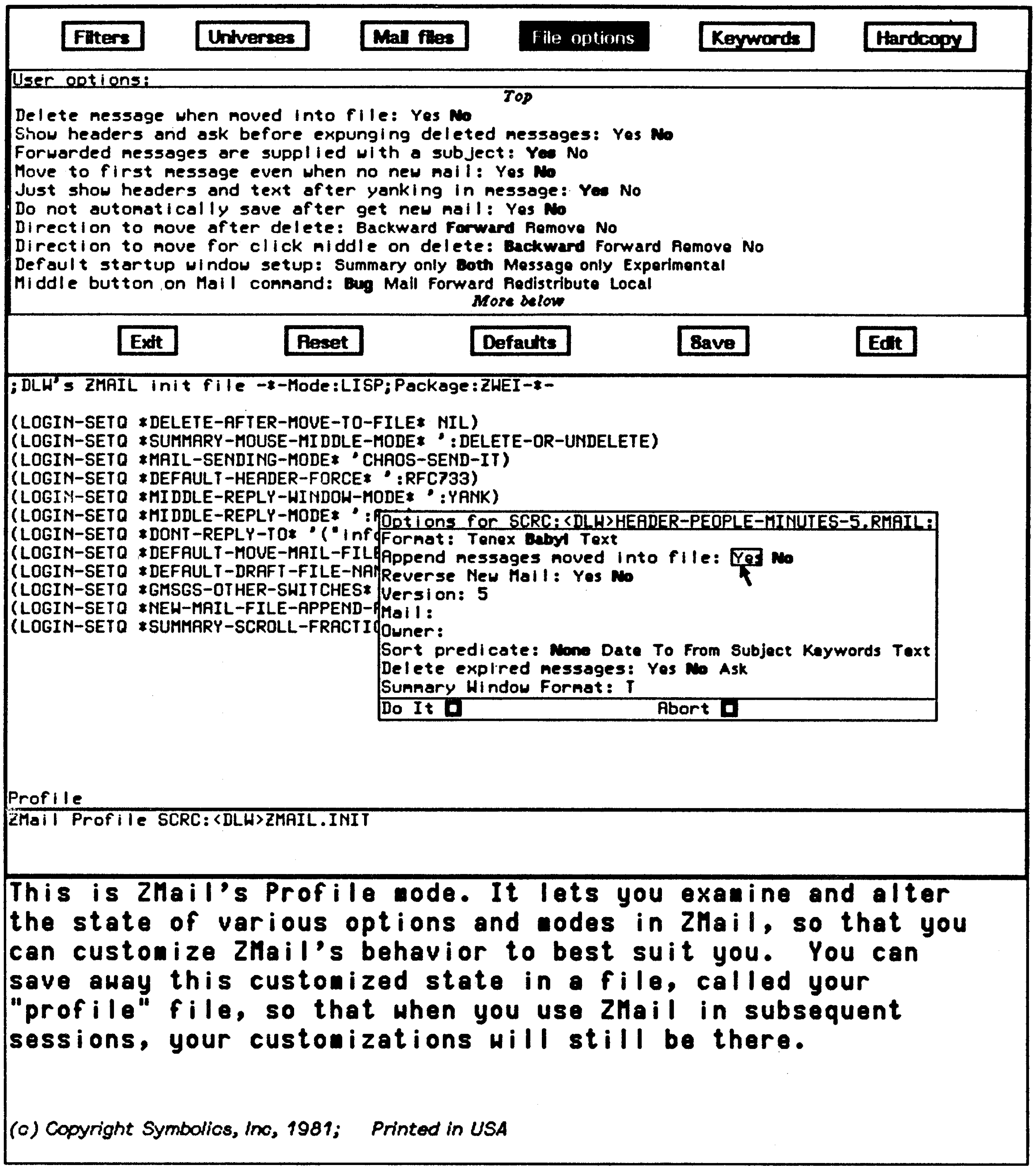
File System Editor

This is the file system editor. The user is examining the files matching the name >Silverstein>math>*.*.* and can now perform any of the listed operations on the file that the mouse was indicating when the menu was called for.



```
 No. Lines  Date From→To                 Subject or Text
 15:     8 23-Feb MARS-RETRIE→Header-Peop Please ignore previous message
 16:    25 24-Feb Feinler@SRI-KL→         Online address for delivery of FIPS standard comments
 17:    25 24-Feb Rick Gumper→PBARAN@USC- net replies
 18:    51  3-Mar Kahler@SUMEX-AIM→       FIPS Proposed Standard Sample Session
 19:    27  3-Mar Feinler@SRI→header-peop FIPS standard
 20:   216  4-Mar RMS@MIT-AI→watkins@NBS- I am not certain that I will be able to find the ment
 21:    53  4-Mar MRC@SU-AI→Feinler@SRI-K FTP in reality and myth
 22:    41  6-Mar MRC@SU-AI→MsgGroup@USC- FTP continued
 23:    17  6-Mar KLH@MIT-MC→mrc@SU-AI    Read RFC 691.  It should be available from SRI-KL as
 24:    12  7-Mar Kahler@SUME→Header-Peop May I be put on the mailing list?
 25:    39  9-Mar JZS@CCA→Kahler@SUMEX-AI Re: May I be put on the mailing list?
 26:    32 10-Mar Dcrocker@Ra→Header-Peop RFC 733 Mail standard:  Group Names
 27:    14 10-Mar Kahler@SUME→Roach@MIT-
 28:   354 10-Mar Kahler@SUME→Watkins@NB                                                 Is
 29:    12 11-Mar Philip.Karl→Dcrocker@R
 30:    71 12-Mar BR10@CMU-10→Watkins@NB
▸ 31:    71 12-Mar BR10@CMU-10→Watkins@NB
 32:    90 13-Mar Frankston@M→Watkins@NB
 33:    51 13-Mar HENDERSON@B→Watkins@NB

Date: Sunday, 12 Mar 1978 1920-EST
From: Brian K. Reid <BR10 at CMU-10A>
Subject: Proposed FIPS standard for User
To:   Watkins at NBS-10, Pyke at NBS-10
CC:   MsgGroup at ISI, Header-People at
      Saltzer at MIT-MULTICS, Tavares at

Associate Director for ADP Standards
Institute for Computer Sciences and Tech
National Bureau of Standards
Washington, DC 20234

Re:     Proposed FIPS entitled "User-Terminal Protocols -- Entry and
        Exit Procedures between Terminal Users and Computer Services"

Sir or Madam:

Message
ZMail SCRC:<DLW>HEADER-PEOPLE-MINUTES-5.RMAIL    Msg #31/364  {}  --More below--
```

This is the ZMail mail reading system. The user is choosing a filter which will select a subset of the messages in the mail file, based on a specific criterion.



```
User options:
                                  Top
Delete message when moved into file: Yes No
Show headers and ask before expunging deleted messages: Yes No
Forwarded messages are supplied with a subject: Yes No
Move to first message even when no new mail: Yes No
Just show headers and text after yanking in message: Yes No
Do not automatically save after get new mail: Yes No
Direction to move after delete: Backward Forward Remove No
Direction to move for click middle on delete: Backward Forward Remove No
Default startup window setup: Summary only Both Message only Experimental
Middle button on Mail command: Bug Mail Forward Redistribute Local
                              More below
```

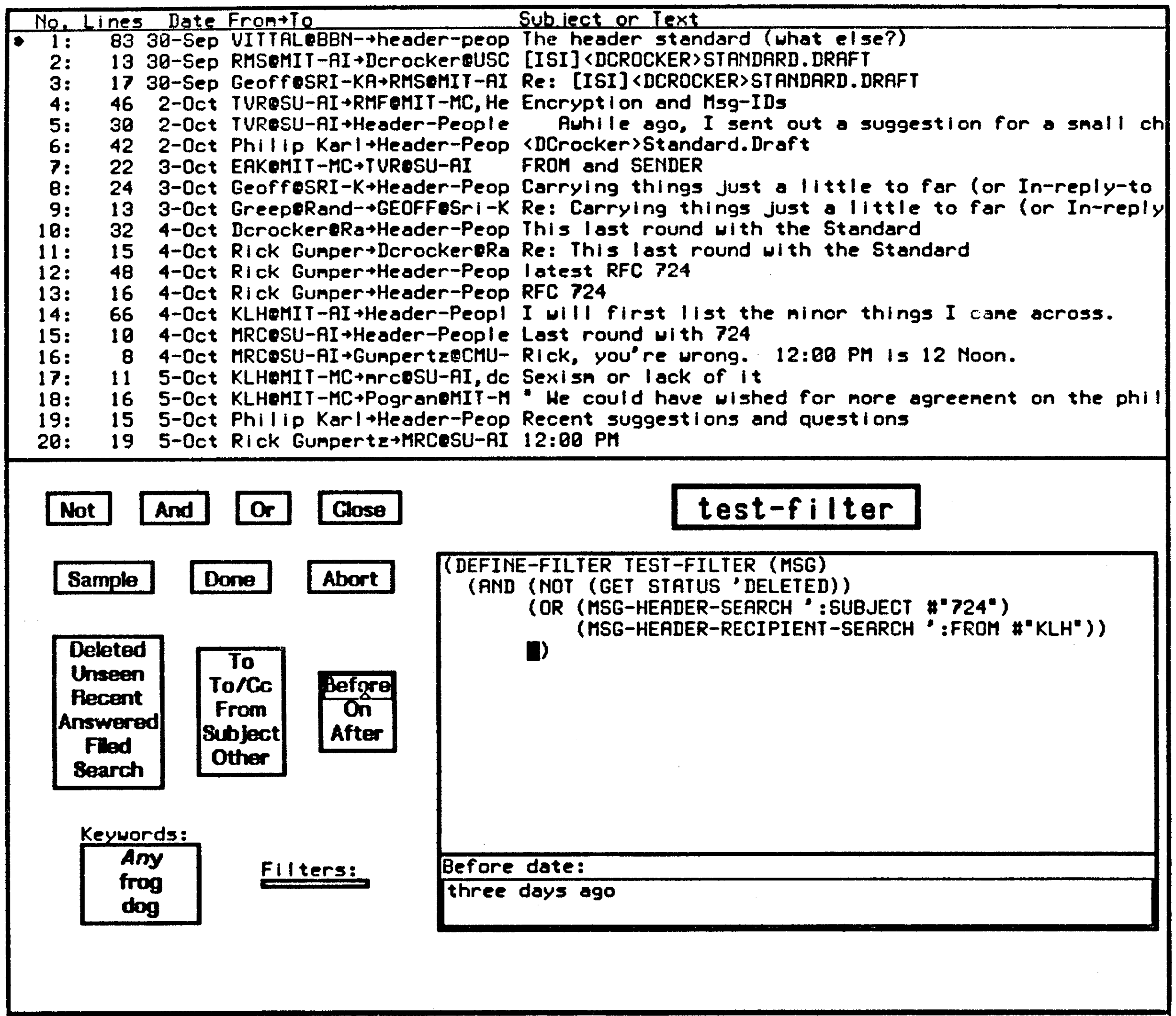
```
;DLW's ZMAIL init file -*-Mode:LISP;Package:ZWEI-*-

(LOGIN-SETQ *DELETE-AFTER-MOVE-TO-FILE* NIL)
(LOGIN-SETQ *SUMMARY-MOUSE-MIDDLE-MODE* ':DELETE-OR-UNDELETE)
(LOGIN-SETQ *MAIL-SENDING-MODE* 'CHAOS-SEND-IT)
(LOGIN-SETQ *DEFAULT-HEADER-FORCE* ':RFC733)
(LOGIN-SETQ *MIDDLE-REPLY-WINDOW-MODE* ':YANK)
(LOGIN-SETQ *MIDDLE-REPLY-MODE* ':
(LOGIN-SETQ *DONT-REPLY-TO* '("inf
(LOGIN-SETQ *DEFAULT-MOVE-MAIL-FIL
(LOGIN-SETQ *DEFAULT-DRAFT-FILE-NA
(LOGIN-SETQ *GMSGS-OTHER-SWITCHES*
(LOGIN-SETQ *NEW-MAIL-FILE-APPEND-
(LOGIN-SETQ *SUMMARY-SCROLL-FRACTI
```

```
Options for SCRC:<DLW>HEADER-PEOPLE-MINUTES-5.RMAIL:
Format: Tenex Babyl Text
Append messages moved into file: Yes No
Reverse New Mail: Yes No
Version: 5
Mail:
Owner:
Sort predicate: None Date To From Subject Keywords Text
Delete expired messages: Yes No Ask
Summary Window Format: T
Do It ☐                    Abort ☐
```

```
Profile
ZMail Profile SCRC:<DLW>ZMAIL.INIT
```

This is ZMail's Profile mode. It lets you examine and alter the state of various options and modes in ZMail, so that you can customize ZMail's behavior to best suit you. You can save away this customized state in a file, called your "profile" file, so that when you use ZMail in subsequent sessions, your customizations will still be there.

(c) Copyright Symbolics, Inc, 1981; Printed in USA

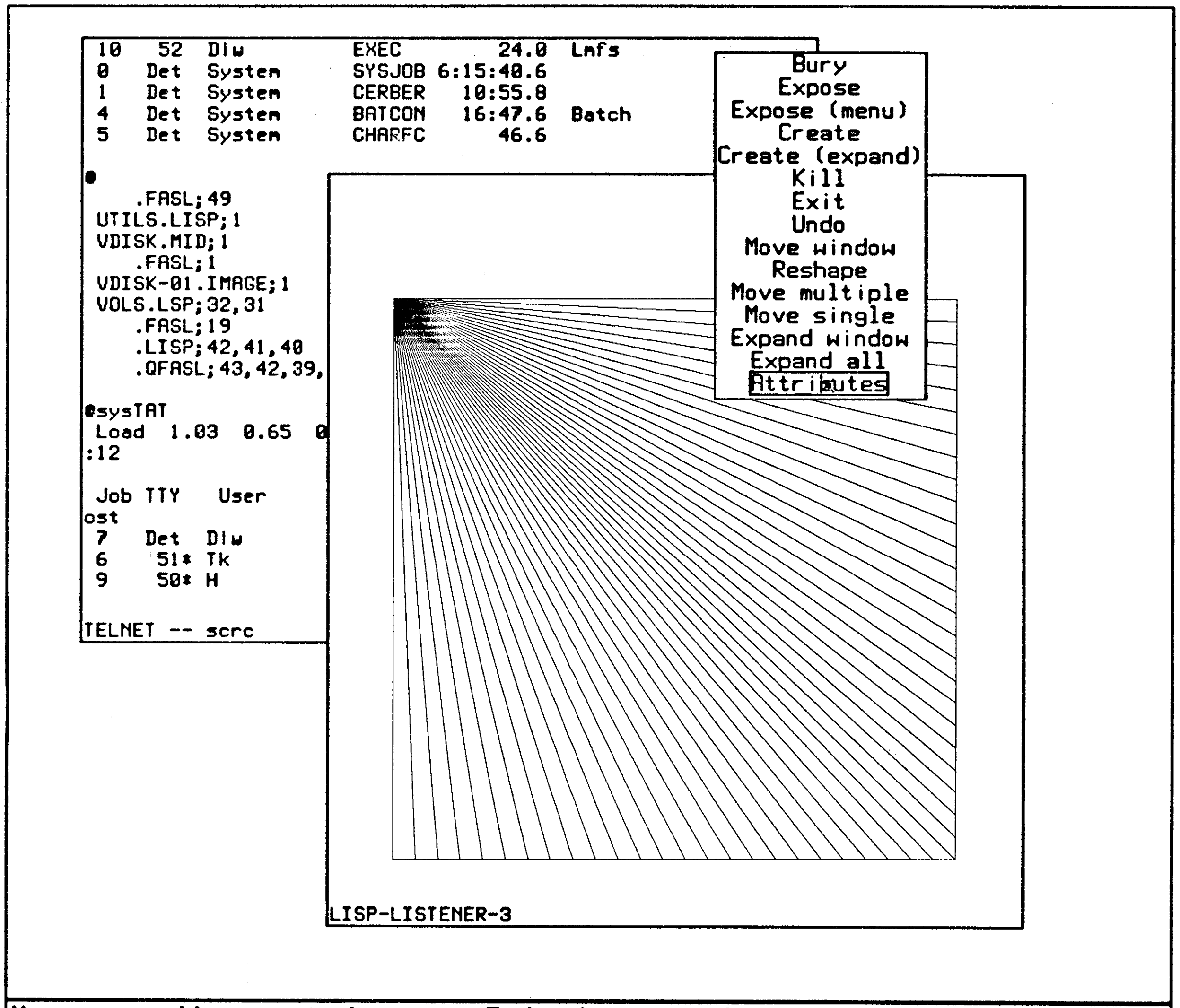


```
 No. Lines  Date From→To                 Subject or Text
▸  1:   83 30-Sep VITTAL@BBN-→header-peop The header standard (what else?)
   2:   13 30-Sep RMS@MIT-AI→Dcrocker@USC [ISI]<DCROCKER>STANDARD.DRAFT
   3:   17 30-Sep Geoff@SRI-KA→RMS@MIT-AI Re: [ISI]<DCROCKER>STANDARD.DRAFT
   4:   46  2-Oct TVR@SU-AI→RMF@MIT-MC,He Encryption and Msg-IDs
   5:   30  2-Oct TVR@SU-AI→Header-People    Awhile ago, I sent out a suggestion for a small ch
   6:   42  2-Oct Philip Karl→Header-Peop <DCrocker>Standard.Draft
   7:   22  3-Oct EAK@MIT-MC→TVR@SU-AI    FROM and SENDER
   8:   24  3-Oct Geoff@SRI-K→Header-Peop Carrying things just a little to far (or In-reply-to
   9:   13  3-Oct Greep@Rand-→GEOFF@Sri-K Re: Carrying things just a little to far (or In-reply
  10:   32  4-Oct Dcrocker@Ra→Header-Peop This last round with the Standard
  11:   15  4-Oct Rick Gumper→Dcrocker@Ra Re: This last round with the Standard
  12:   48  4-Oct Rick Gumper→Header-Peop latest RFC 724
  13:   16  4-Oct Rick Gumper→Header-Peop RFC 724
  14:   66  4-Oct KLH@MIT-AI→Header-Peopl I will first list the minor things I came across.
  15:   10  4-Oct MRC@SU-AI→Header-People Last round with 724
  16:    8  4-Oct MRC@SU-AI→Gumpertz@CMU- Rick, you're wrong.  12:00 PM is 12 Noon.
  17:   11  5-Oct KLH@MIT-MC→mrc@SU-AI,dc Sexism or lack of it
  18:   16  5-Oct KLH@MIT-MC→Pogran@MIT-M " We could have wished for more agreement on the phil
  19:   15  5-Oct Philip Karl→Header-Peop Recent suggestions and questions
  20:   19  5-Oct Rick Gumpertz→MRC@SU-AI 12:00 PM
```

```
(DEFINE-FILTER TEST-FILTER (MSG)
  (AND (NOT (GET STATUS 'DELETED))
       (OR (MSG-HEADER-SEARCH ':SUBJECT #"724")
           (MSG-HEADER-RECIPIENT-SEARCH ':FROM #"KLH"))
       )
```

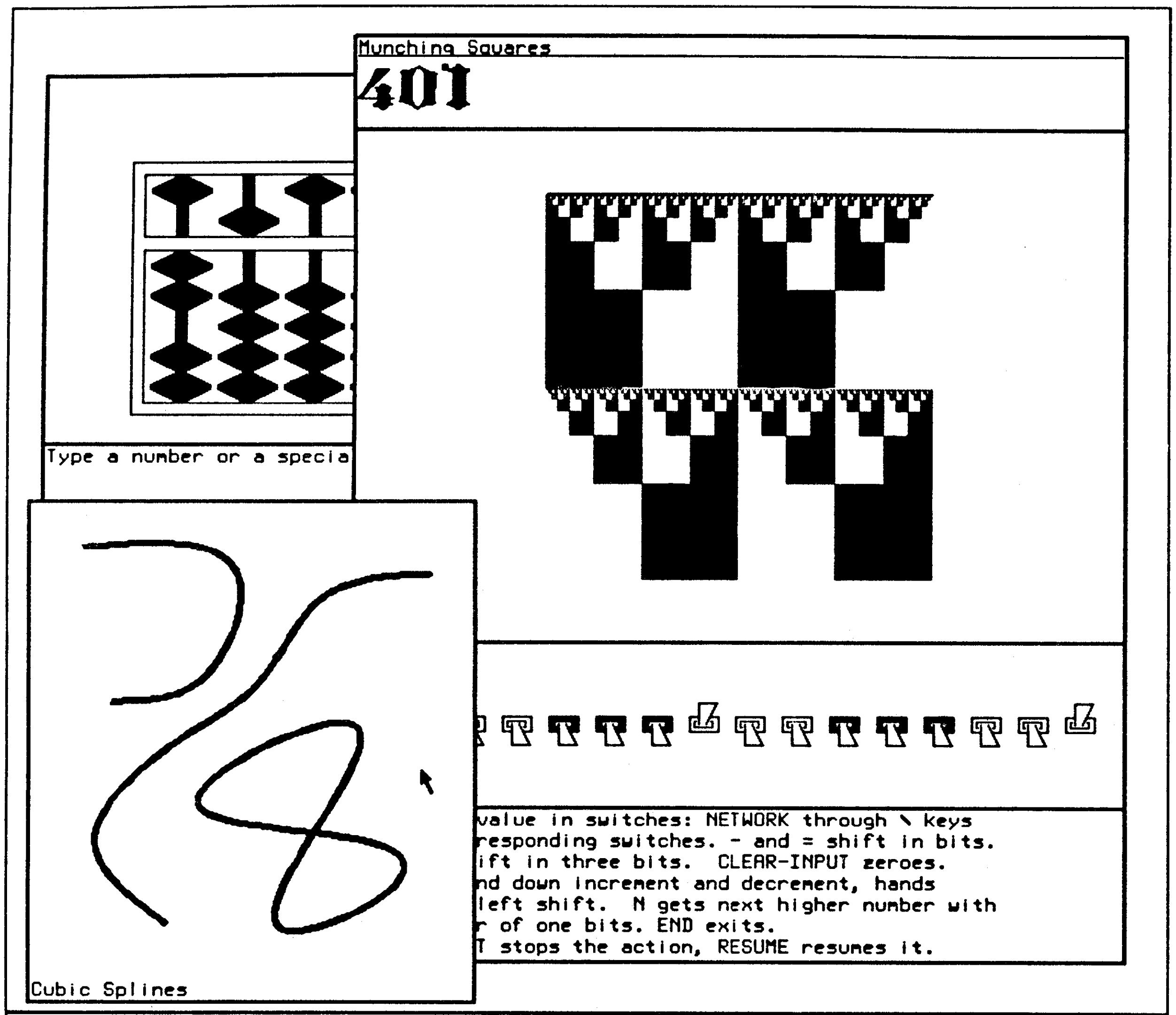
```
Before date:
three days ago
```

This is ZMail's filter definition mode. This filter finds all messages that have not been deleted, and that either contain "724" in their subject lines or are from "KLH", and were sent more than three days ago.



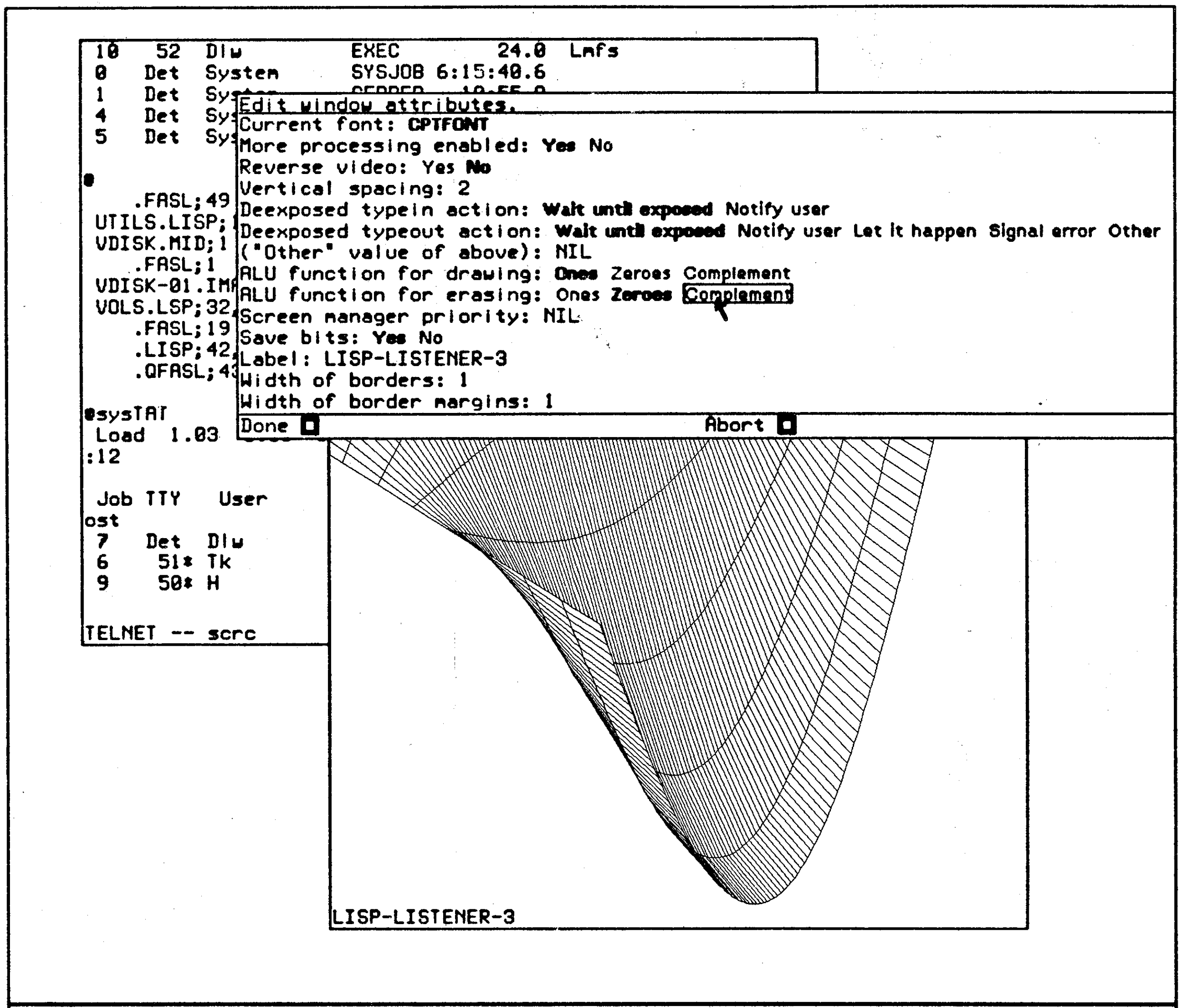
```
10   52  Dlw         EXEC        24.0  Lmfs
0    Det  System     SYSJOB 6:15:40.6
1    Det  System     CERBER   10:55.8
4    Det  System     BATCON   16:47.6  Batch
5    Det  System     CHARFC      46.6

●
    .FASL;49
 UTILS.LISP;1
 VDISK.MID;1
    .FASL;1
 VDISK-01.IMAGE;1
 VOLS.LSP;32,31
    .FASL;19
    .LISP;42,41,40
    .QFASL;43,42,39,

@sysTAT
 Load  1.03  0.65  0
:12

 Job TTY   User
ost
 7   Det  Dlw
 6    51* Tk
 9    50* H

TELNET -- scrc
```

Here are three windows: a Telnet connection, a Lisp Listener showing a graphics display, and a Screen Editor command menu.



```
value in switches: NETWORK through \ keys
responding switches. - and = shift in bits.
ift in three bits.  CLEAR-INPUT zeroes.
nd down increment and decrement, hands
left shift.  N gets next higher number with
r of one bits. END exits.
T stops the action, RESUME resumes it.
```

These are some examples of the use of graphics displays.



```
 10   52  Diw         EXEC        24.0  Lmfs
 0    Det  System     SYSJOB 6:15:40.6
 1    Det  Sy
 4    Det  Sy
 5    Det  Sy

    .FASL;49
 UTILS.LISP;
 VDISK.MID;1
    .FASL;1
 VDISK-01.IM
 VOLS.LSP;32
    .FASL;19
    .LISP;42
    .QFASL;4

@sysTAT
 Load  1.03
:12

 Job TTY   User
ost
 7   Det  Diw
 6    51* Tk
 9    50* H

TELNET -- scrc
```

```
Edit window attributes.
Current font: CPTFONT
More processing enabled: [✓ Yes] No
Reverse video: Yes [✓ No]
Vertical spacing: 2
Deexposed typein action: [✓ Wait until exposed] Notify user
Deexposed typeout action: [✓ Wait until exposed] Notify user Let it happen Signal error Other
("Other" value of above): NIL
ALU function for drawing: [✓ Ones] Zeroes Complement
ALU function for erasing: Ones [✓ Zeroes] Complement
Screen manager priority: NIL
Save bits: [✓ Yes] No
Label: LISP-LISTENER-3
Width of borders: 1
Width of border margins: 1
Done □                                   Abort □
```

LISP-LISTENER-3

This Choose Variable Values window was created by the Attributes command of the Screen Editor. It lets the user examine and alter various attributes of the window called LISP-LISTENER-4.