Displaying Data Structures for Interactive Debugging

by Brad A. Myers

CSL-80-7 June 1980

Abstract: See next page

This report reproduces a thesis submitted to the Department of Computer Science and Electrical Engineering of the Massachusetts Institute of Technology in partial fulfillment of the requirements for the degrees of Bachelor of Science and Master of Science in Computer Science.

CR Categories: 4.29, 4.34, 4.41, 4.42, 8.2.

Key words and phrases: Debugging, data structures, user interface, symbol tables, abstract data types, graphics.

XEROX
PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California 94304

DISPLAYING DATA STRUCTURES FOR INTERACTIVE DEBUGGING

by

BRAD ALLAN MYERS

ABSTRACT

Many modern computer languages have a variety of basic data types and allow the programmer to define more. The facilities for debugging programs written in these languages, however, seldom provide any capabilities to capture the abstraction represented in the programmer's mind by the data types. *Incense*, the system described here, is a working prototype system that allows the programmer to interactively investigate data structures in programs. The desired displays can be specified by the programmer or a default can be used. The defaults include using the standard form for literals of the basic types, the actual names for enumerated types, stacked boxes for records, and curved lines with arrowheads for pointers. The intention is that the display produced should be similar to the picture the programmer would have drawn to explain the data type. Incense displays have the additional feature that they can change dynamically.

Incense is written in and for the Pascal-like language *Mesa*, which was developed at the Xerox Palo Alto Research Center. Incense has been used to investigate and document many data structures including some of the internal data structures of the Incense system itself.

In addition to displaying data structures, Incense also allows the user to select, move, erase and redimension the resulting displays. Incense also allows the user to modify the actual values stored using the same high-level names that are displayed. These functions are provided in a uniform, natural manner using a pointing device ("mouse") and keyboard.

ACKNOWLEDGMENTS

I would like to thank my advisor at MIT, David Reed, and all the people at PARC who made this effort possible. Dan Swinehart, my advisor, was available for helpful discussions and design suggestions and even read almost all of my voluminous memos. John Warnock, who wrote the underlying graphics system as well as the interpreter I used to debug Incense, was exceedingly helpful in explaining how things worked and adding needed features quickly. Ed Satterthwaite was gracious enough to answer my repeated cries for help when I was trying to write a run-time type system using the compiler's symbol tables. I would also like to thank the many other people who contributed to the design of Incense including Butler Lampson, Warren Teitelman, Bill Paxton and Paul Rovner. Many others also helped by answering my questionnaires or commenting during demonstrations.

Special thanks must go to Dan Swinehart, Warren Teitelman, Ed Satterthwaite, David Reed and many others for their helpful and thorough comments on earlier versions of this paper.

I would like to thank Mike Schroeder, John Tucker and the MIT 6-A program for giving me the wonderful opportunity to work for Xerox PARC and to Bob Taylor and CSL for providing the facilities and support for Incense. The generosity of Xerox in giving to MIT some Alto computers allowed me to continue work on this paper even after leaving PARC, for which I am grateful. I am also indebted to the staff of PARC's Technical Resource Center who helped immensely while I was doing research for this paper.

Finally, I would like to express my appreciation to my parents and grandparents for making it all possible and to my sisters for their continued support.

TABLE OF CONTENTS

LIST OF FIGURES

I. Introduction

Many modern computer languages have a variety of basic data types and allow the programmer to define others. Few languages, however, have facilities to allow the programmer to display these data structures for debugging, monitoring or documenting programs in a reasonable fashion. This thesis describes the system *Incense*, written in the Mesa computer language [Mitchell 79b], which allows the client to design and use graphical representations for data structures.

Pictures are clearly useful for representing data since they are used by programmers to explain their data structures to other humans. Frequently, a picture will be drawn of the typical case or the one under examination. A system that could present the information in the same manner that a programmer does would thus be taking a large step towards making the information easier to understand.

The most basic part of Incense is simply a framework for data display. Thus, Incense would be useful to many types of systems that have data display as a component. The major emphasis of the current work, however, has been in the area of debugging systems. Incense has therefore been augmented with a large number of procedures that automatically display the data structures found in actual Mesa programs. In addition, facilities exist to allow the user to specify and modify the displays at various levels.

The most difficult aspects of Incense were the display of pointers, and allowing the client to define new display formats. The solution to the first problem involved using a carefully designed abstraction to hide the internal data and procedures. The problem with pointers is that a location on the screen must be chosen for the referent. Incense provides a mechanism to allow this to be done without dynamic space allocation. Although presently Incense does not have an acceptable front end, it should increase the effectiveness of any debugger into which it might be integrated.

1.1 Importance of Debugging.

Relatively little work has been done on debuggers and data structure display since the early days of computer science. Model [79, p. 4] claims that "the past twenty years has seen little change in the nature of the debugging facilities available to the typical programmer." The debuggers for high level languages such as Fortran, Pascal, and Mesa have mostly copied the aids developed for assembly languages. One reason for this is that programmers tend to discount the importance of debugging. Some, such as Dijkstra [72, p. 863], claim that good programmers should not waste their time debugging because "they should not introduce bugs to start with." When interesting debugging facilities are developed, they are frequently not documented or published since the

systems are frequently proprietary and specific to a particular machine and/or language. In the area of data structure display, there has been even less work. In fact, "few attempts have been made to create formal external representations for the data environments (for any language)" [Swinehart 74, p. 83].

Unfortunately, the problem of debugging is not likely to go away. Even with the use of modern languages and the advent of structured programming techniques, programmers still spend "countless hours" debugging [Hanson 78]. Van Tassel [74, p. 117] goes so far as to claim that "a bug-free program is an abstract theoretical concept." Naur [74, p. 54] substantiates this claim: "The difficulty in achieving correctness in programs may be understood when the degree of complexity of many programs and the need for virtually absolute correctness of all details of these programs is considered."

The importance of debugging can also be seen in its costs. Estimates of the amounts of time programmers spend debugging vary from fifty to ninety percent of the programming task [van Tassel 74, p. 117]. Debugging and maintenance together may cost fifty times more than original program production [Tratner 79, p. 97].

1.2 The Theory and Teaching of Debugging.

Unfortunately, few theories exist about how people debug and what makes a debugging system more effective. Debugging is generally considered an art, a creative activity. In finding the errors in a program, "the programmer operates in a very intuitive mode, depending more on insight and imagination than on rigorous step-by-step analysis" [Model 79, p. 53]. Some studies have attempted to classify types of bugs [van Tassel 77] and to find some global rules to help programmers (*e.g.*, [Loeser 76]). Atwood [78] did a study based on cognitive psychology to try to develop a theory of the difficulty of finding different types of bugs. He claims that the deeper the logical nesting depth of a bug's location in the program, the harder it is to find. Sheppard [79] presents evidence, however, that Atwood's findings might not be valid and that there are strong dependencies on the particular algorithm used in the study. Levine [77] attempted to discover if carefully planned debugging was more effective than "ad hoc" techniques but found no significant results. Most of these studies have used small programs with very small numbers of subjects (*e.g.*, 10) so no real conclusions can be drawn from them.

One result of the lack of a theory of debugging is that it is difficult to teach debugging techniques. "From the beginning," Tratner [79, p. 97] claims, "our best schooling in debugging teaches futility." All that texts and teachers can do is suggest general approaches and demonstrate the debugging aids that are available.

1.3 Importance of Monitoring.

In many cases, it is important to be able to monitor the state of a program while it is executing. A graphical display, such as presented by Incense, would allow the user to more easily follow the monitoring process (see chapter 2). Frequently, the information needed to locate an error is no longer available when recognized as important. This is especially true with real-time programs such as operating systems. In an informal poll of Mesa users (see Appendix A), two of the most frequently listed problems were random overwriting of memory locations and process interactions (including timing problems). Monitoring that allowed watching of both control flow and variables would help immensely in locating the sources of these bugs.

The programmer may also be able to discover problems with the flow of control or manipulations of the data structures using monitoring. Swinehart [74, p. 2] explains: "Continuous display of information with some associated context helps the user to retain comprehension of complex program environments, and to indicate the environments to be affected by his commands."

1.4 Overview of Thesis.

In view of the importance of debugging and the general lack of research and theories about it, the best way to investigate ideas about better techniques seems to be to implement a system and then see if it appears to increase the programmer's productivity. Incense is an attempt to study one very important part of debugging systems: the display of data structures. This thesis first presents some of the requirements felt to be important in any debugging system (chapter II) and then describes some related work in debuggers and graphical systems (chapter III). Following this is a description of the environment at the Palo Alto Research Center (PARC) in which Incense was created (chapter IV). An overview of the Incense system (chapter V) is followed by a discussion of the run time type system required for generating the default displays (chapter VI). More detail about the actual implementation of Incense is then given (chapter VII). The thesis concludes with some ideas for future work (chapter VII) and a summary and conclusion (chapter VIII). It is important first, however, to define the terms used in the rest of the paper.

1.5 Definition of Important Terms.

This paper assumes the reader is familiar with programming and some computer languages; however, some terms will be used in a specific or unusual manner and are therefore defined below.

Data. Knuth, in his famous text book [Knuth 69, p. 620], defines data as

> representation in a precise, formalized language of some facts or concepts, often numeric or alphabetic values, in a manner which can be manipulated by a computational method.

Data Types. All variables in languages such as Mesa are required to be of some specific *type.*

> Each type is characterized by (1) the set of values included in it, (2) the way literals of that type can be written in a program, (3) rules about how the values of the type may be used as operands of operations, and (4) rules about how values of the type result from executing operations. [Naur 74, p. 40]

A type may also specify the way the values are to be laid out in memory [Aho 78, p. 387]. Types in Mesa include INTEGER, REAL, CARDINAL (positive integers), BOOLEAN (true or false), CHARACTER, POINTER, RECORD, ARRAY, ENUMERATED (lists of names, e.g. {Mon, Tues, Wed, Thurs, Fri}), subranges of the above types, and many other more esoteric types (see chapter 6).

Strongly Typed Languages. Strongly typed languages are ones "that have many types, but the type of every name and expression must be calculable at compile time" [Aho 78, p. 36]. Mesa and Pascal are strongly typed, but Pl/1, Fortran and Lisp are not. This feature provides many opportunities for a data display system since the compiler can save the type information allowing the correct display format of any data to be chosen.

Data Structures. "A data structure is a set of primitive data elements and other data structures, together with a set of structural relations among its components" [Aho 78, p. 38]. Thus instances of records and arrays are data structures under this definition. In this thesis, however, I will frequently use *data structure* to also include the basic data elements.

Debugging. Debugging "is the process of making a program behave as intended. The difference between the intended behavior and actual behavior is caused by 'bugs' (program errors) which are to be corrected during debugging" [Lauesen 79, p. 51]. Model [79, p.52] classifies different stages of debugging:

> Fundamentally, debugging is the act of (1) observing the behavior of a computer program; ... (2) comparing the actual behavior to the behavior desired of the program; (3) analyzing the cause of variances thereby detected; (4) devising changes to the program that would make it conform more closely to the desired behavior; and (5) altering the program in accordance with those changes. Normally the act is cyclical: after the program has been modified, the steps are repeated until a sufficient match between desired and observed behavior is obtained.

Testing. Some authors distinguish between testing and debugging. Van Tassel [74, p. 118] asserts that "testing determines that an error exists; debugging localizes the cause of the error." Debugging will be used to include testing in this thesis, however.

Client. In the text above, there have been references to what the programmer can do in Incense. In fact, however, the system is configured so that a program can call on Incense and get the same results. *Client* will be used to denote both human and procedural users, and *programmer* and *user* will be used interchangeably to refer exclusively to the human user.

II. Desired Features in a Debugging System

Many different requirements for debugging systems have been specified in the literature, and actual systems conform to these constraints to varying extents. This section will attempt to list many of the desired features of debugging and data display systems and the reasons they are appropriate. These requirements will be used as criteria for judging debugging systems and will serve as goals for Incense.

2.1 Motivation for Features.

As might be expected, most of the desired features are motivated by limitations of the human users. People can only attend to a small amount of information at a time, but "when dealing with information in a familiar form, ... humans are highly adaptable. They tend to supply missing items themselves" [Naur 74, p. 238]. Another aspect of human understanding is that it tends to be highly "context sensitive" [Swinehart 74, p. 10]. That is, the presentation of context allows more rapid recognition of the information's meaning and significance.

Another aspect of the programmer-computer interface is the volume of data that needs to be transferred. Frequently, the programmer must process large amounts of information produced by the computer, for example to find a bug with unusual, non-local effects. Furthermore, this data may be generated by multiple processes running in the computer. In this case, the programmer must be able to separate the output coming from the various sources.

2.2 Features.

Naur [74, pp. 239-245] gives a long list of design requirements for any computer system for which a human interface must be provided. The principles governing design of input from the user are:

(1) Reduce the volume of data required from the user;
(2) Adjust the form of the input to make best use of the human faculties;
(3) Use feedback to allow correction of mistakes as they are made; and
(4) Include careful redundancy to allow automatic recognition of mistakes [Naur 74, p. 244].

For output, Naur has a similar list:

(5) Adjust output quantity to human capacity;
(6) Choose forms of output that are readily acceptable to human comprehension; and
(7) Output only completely processed results [Naur 74, p. 245].

This section will discuss these points (and some others) emphasizing their applicability to debugging systems.

2.2.1 Speed.

One of the most common complaints about current systems is that the operations required to find a bug take too long. Many of the experimental systems that seemed very attractive on paper (for example, [Model 79]), are not used because they operate too slowly [Fikes 79][Laaser 79]. Systems such as RAID [Petit 69], which essentially add a Cathode Ray Tube (CRT) screen to conventional debugging aids, are very effective partially because "relatively conventional tools are considerably enhanced by increasing the bandwidth of the communication path" [Satterthwaite 75, p. 21].

Mitchell [79a, p. 7] gives a more theoretical definition of the speed requirements of an interactive system. He proposes that the two conversants should be matched in response time (a "balanced conversation"). If not, the more powerful one will either use his time poorly or waste energy doing unnecessary computation. Different types of tasks are allowed to take different amounts of time, however. For example, the wait before commencing a complex operation and seeing it complete may be much longer than the wait between pressing of a key and seeing the letter typed. Recently, it has become acceptable to allow the computer to be under-utilized due to decreasing costs of hardware. Consequently, the appropriate criterion today is that the human should be able to avoid wasting his time.

2.2.2 Information at user's level.

Naur requires that output should be "completely processed." For example, if the user requests the display of a CHARACTER and the debugger prints an octal number, the user rather than the computer will have to do the conversion. A similar requirement also applies to input. With respect to programming languages, "it is imperative that information about the behavior of a program be presented at the conceptual level at which the program was written and in terms of the constraints and operations of the programming language used" [Model 79, p. 55].

A motivation for this principle is given by Model [79, p. 28]. He explains that people do not usually understand higher level constructs by breaking them down into the lower level constituents because these details require the user to handle more information, most of which is irrelevant to his task. If the information is presented at a higher level, this "leaves more of the programmer's resources available for the demanding analytic and creative phases of debugging activity" [Model 79, p. 55].

Another aspect of this requirement is that the programmer can best understand the information if the context in which the information is to be evaluated is presented. Most current systems require the user to declare the context in which he is interested. The system then assumes

he remembers it. In systems with complex, dynamic displays, however, sufficient contextual information must be displayed to allow the programmer to identify the meaning of the displayed information.

2.2.3 Use of appropriate level of detail.

In addition to the requirement that the information be presented at the correct conceptual level, the amount of information produced should be minimized. If the user is interested in one particular value, he should not be required to hunt through a large amount of output to find it. "The name of an object is often all the programmer needs to see, as printouts frequently serve only to *identify which one* of a set of known objects a particular one is. When the user does want further details, it is imperative that he be able to request them selectively" [Model 79, p. 78]. Providing only the needed information can also allow the system to respond much more quickly since interactive systems frequently spend much time handling Input/Output (I/O).

2.2.4 Analogical display.

The use of *analogical* display helps to handle many of the requirements listed above. An analogical display uses abstract pictures, such as bar graphs, icons, arrows, tables, etc. Thus the information is not simply printed out; it is converted into a form that is easier for the human to understand, possibly by *analogy* to the physical world.

Again, Model [79, p. 12] motivates this requirement with reference to human psychology. The structure of the human brain and current understanding of human information processing "show that sensory information [of the physical world] is highly organized before it reaches the parts of the brain associated with abstraction, analysis, and other components of thought." Thus, analogical displays more effectively utilize the brain's innate capacities. For example, the eye is good at making rough estimates in proportion. A display of an iteration variable as a "percent-done thermometer" (like those used in charity drives; see Figure 2.1) or a bar graph (Figure 2.2) may present all of the required information in a manner that is easy to understand.

Pictures are used by programmers to explain their data structures to other humans. For example, the programmer often draws a picture of the typical case or the one under examination. The programmer rarely would write a set of octal numbers to explain how his tree-structure is represented. Therefore, a system that uses pictorial output presents the information in a more natural and understandable manner.

Figure 2.1. Percent-done thermometer showing 80% complete.

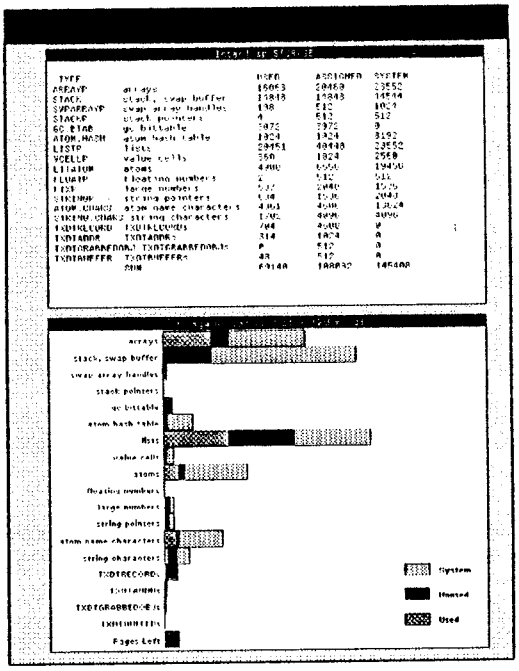


Figure 2.2. Display of storage usage in DLISP as numbers (top) and bar-graph (bottom). [Model 79, p. 117].

2.2.5 Automatically generated pictures.

Pictures are also produced as documentation for programs. The most obvious example of this is flowcharts. Drawings of data structures are also frequently drawn by hand, especially for languages with standard, high-level structures such as LISP. Unfortunately, "readable, neat copies of pictures are expensive to produce and reproduce ... [and] with manually generated pictures, there is no guarantee of correspondence with a real computation running on a real machine" [Yarwood

77, p. 9]. Thus, automatic generation of the pictures is required. This was done on the earliest computers (section 3.6.1), but was seldom used until recently (section 3.5.2). Interest in automatic generation of flowcharts began in the late 1950's (see section 3.6). The automatic generation of pictures as a means of display seems the natural choice considering the desire for analogical display.

2.2.6 Meta-knowledge.

Debugging systems might help the programmer understand the output at a more global level even more than by using analogical display. For example, when asked the value of an array, the system might say: "All have initial value except for element #17 which equals ...." This would usually be easier to understand than the list of all the values. Another useful facility would be to detect that certain elements of a complex structure had been overwritten by accident. At an even higher level, the programmer might ask, "Did anything unusual happen during this execution?" A system that would allow expression of these sorts of requests would require knowledge and capabilities not yet available even in the most advanced artificial intelligence systems. If the techniques were available, however, debugging systems would be a useful place for them to be applied replacing much of the current interface to debuggers. A sophisticated data display system would still be useful, however, and would be improved through the use of knowledge about the user and program.

2.2.7 Replay.

The ability to make a history of the events that occurred during an execution or debugging session has proved useful in many systems. If the running time of the program under investigation is too long, a replay at a higher speed using the history may be the only practical way of following it [Lenders 78]. Replays may also allow the user to back up a computation, change something, and then repeat the computation in the new environment. Yarwood [77, p. 61] gives another motivation for replays:

> In order to understand a complex change in ... data, the user must be able, for a short time at least, to move his attention more or less randomly between the "before" state, the "after" state, and the intervening part of the program which caused the change.

In a CRT based environment, a history should be stored to allow the user to look back at previous output. With an analogical display, the information stored on the history should be sufficiently detailed to allow a replay of the pictures produced originally.

III. History of Debuggers and Other Relevant Systems

Debugging of computer programs has been necessary ever since the first programs were written. There are many good histories and surveys of debugging (for example, [Model 79], [Satterthwaite 75], and [Blair 71] all contain good surveys over different sets of systems). The survey presented here makes no attempt to be thorough or comprehensive. The important stages of computer software debugging are presented along with some illustrative examples in sections 3.1 through 3.5.

Some systems that were not aimed directly at debugging have profoundly affected the way humans interact with the computers. These have generally been graphically oriented. and some are discussed in section 3.6.

3.1 Earliest Systems and the Basic Debugging Techniques.

The earliest stored program computers were small enough that one user's program could effectively utilize the computer's resources. Therefore, the programmer could sit at the console and debug his program while it was operating. This is called the *interactive* mode of operating since the user is interacting with the computer. The alternative is *batch* mode, in which the user has no control over the execution of the program once it has begun. The earliest interactive debugging was frequently done by watching the lights on the front panel of the computer. The three basic forms of debugging, *trace, dump*, and *break*, were all developed on the EDSAC, the "first practical stored-program electronic digital computer" [Satterthwaite 75, p. 18]. The EDSAC was built at Cambridge University in the middle 1940's [Bell 71, p. 42].

3.1.1 The Trace.

In a program *trace*, some portion of the state of the machine, such as the location in the code and the values of important variables, is printed out every time certain events occur. These events are usually the reading or writing of a memory location or the execution of a certain type of instruction (e.g. a branch at a specific point). One purpose of a trace is to "give the user some picture of which of the many possible sequences of operations was actually performed" [Model 79, p. 44]. Flow tracing is useful for optimizing code, since the user can discover where the program is spending its time. Tracing is also useful for discovering how some variable got a certain value.

3.1.2 The Dump.

The *dump* is actually a more primitive operation than a trace. The programmer displays all the values in a certain area of memory, usually in some numerical representation such as octal. He

must then try to figure out what all the values mean and if any of them are wrong. Dumps are inefficient since (1) the bug may have occurred long before the effects seen in the dump; (2) finding more than one bug is difficult; (3) too much information is available, making it hard to find the important parts; and (4) it is hard to get any meaning from a dump if a higher level language is in use [Lauesen 79, p. 53]. Dumps were discouraged on the earlier machines such as the EDSAC, due to the slow output devices available [Bell 71, p. 139].

3.1.3 The Breakpoint.

Whereas traces and dumps can be used in either interactive or batch modes, breakpoints are only useful in the former. A *breakpoint* is simply a method for causing the program to cease executing, usually in a manner that will allow it to resume at the user's command. The standard implementation method is to save the instruction where a break is desired and store a trap instruction in that location [Hughes 78, p. 102]. Care must be taken in executing the instruction out of line to maintain the correct semantics.

Breakpoints only allow "snapshots" of the program state and are not very good for finding certain types of bugs. For example, to find out how code is being clobbered, the programmer might have to repeatedly run his program setting breaks further and further back [Leslie 78]. Breakpoints also generally give confusing information in systems with multiple processes.

A small enhancement on breakpoints is *single-stepping* where the system sets a breakpoint before every instruction. Each system that provides this facility must therefore define the meaning of "an instruction." For example, in LISP, an instruction might be the processing of one atom, the call of a function, or one cycle of the read-eval-print loop.

3.1.4 Events.

Events in a program execution are the occurrences of certain happenings, for examples, accessing or writing of a memory location or the execution of a certain type of instruction. Hanson [78] has formalized the idea of events and provided facilities in the language system SNOBOL for executing arbitrary functions when they occur. He classifies the events into five types: (1) referencing of a variable; (2) execution of a statement; (3) external interruption (by the user); (4) function call or return; and (5) execution time error [p. 116]. A new function has been added to the language definition of SNOBOL called CONNECT which attaches a function to a particular instance of an event type. This mechanism is sufficiently general to allow any type of breakpoint, trace or dump to be written, and it is possible to provide an entire debugging system such as DDT (section 3.4.2).

Hanson claims that the overhead for all of this flexibility is small. One extra compare is required for every assignment to a variable, but this can be omitted if no events associated with variables are used [p. 125]. Thus event associations seem to be a general and powerful way to achieve some of the facilities desired in a debugging system.

3.2 Batch Debugging Systems.

When computers became faster and more expensive, one person could no longer efficiently use the entire machine. Thus, batch processing was invented [Mitchell 79a, p. 4]. Here the programmer develops his program off-line and then (usually) punches it on cards. These are submitted and later its output is available. The turn-around time for batch systems is typically on the order of six hours or more. The programmer was therefore encouraged to carefully examine his program to try to avoid extra runs. Lots of output was generated on each run so the programmer would have some hint as to where bugs might be.

It was in batch processing that dumps became a dominant form of debugging. With the advent of high speed line printers, dumps were much more practical than allowing the user to investigate a running program. Operating systems were frequently configured to dump all of memory when some faults occurred. Lyon [78, p. 1] reports that "whole books have been devoted to deciphering COBOL dumps."

3.2.1 Print statements as a debugging tool.

Unfortunately, dumps frequently proved inadequate and many systems did not have usable trace facilities, so another popular debugging technique emerged. The programmer would insert print statements in his program to try to simulate an effective trace. Knuth [69, p. 189] reports that "many of today's best programmers will devote nearly half of their program to facilitating the debugging process on the other half; ... the net result is a surprising gain in productivity." Problems with this mode of debugging are that the "debugging statements usually must be left out of the final version of the program, and are tedious to include in [its] development" [Hanson 78, p. 121]. Another problem is that the output (or its absence) may modify the program's behavior (*e.g.*, its timing characteristics) in a way that creates or hides bugs.

Lauesen [75] [79] claims that in spite of these problems, he was able to produce an operating system that "seems to be error-free" using print statement output as the major debugging technique [Lauesen 75, p. 378]. He further maintains that "if the program is properly structured, [putting print statements] in a few places will suffice, even in large programs" [Lauesen 79, p. 53]. The output is directed at a file during production runs to act as a history, allowing the programmers to do a mental replay of events if a bug occurs.

3.2.2 An advanced batch system.

Satterthwaite [75] developed a system to aid in the debugging of programs written in Algol W at Stanford University. His system allows the user to add ASSERT statements to the program that the system will then check. If they are not true, the program will be halted, and, as for any other run-time errors, a *post-mortem dump* will be generated containing the values of all variables. A tracing facility was also provided along with a simulator to execute the traced statements. All output was given in terms of the original program and contained only symbolic names for variables and their values.

3.3 Intermediate Debugging Systems.

Some debuggers attempted to handle both batch and interactive access. For example, the PEBUG system is a low-level system that "provides the general debugging environment for the debugging of any relocatable object program" through either batch or interactive interfaces [Blair 71, p. 1]. This system attempts to avoid language dependencies by having a standardized symbol table format. It also allows users to define debugging routines and can tolerate bugs in them.

3.4 Interactive Debugging Systems.

Interactive systems developed along with timesharing in the middle 1960's. Users could now interact with their programs as if they had an entire system to themselves. Later, when hardware became even cheaper, personal computers began to be popular. In these systems, the user actually do have the entire machine. Since the abstraction presented to the user does not differ substantially in the two worlds, they will be discussed together in this section.

Mitchell [79a] describes how an interactive system might be built and provides some insight into the motivation and requirements for one. For example, he claims that "it is a generally held belief that interactive systems should give 'immediate' response to trivial requests" [p. 6]. Examples of interactive programming systems are JOSS, Basic, APL, LCC, Interlisp and COPILOT [Swinehart 74, p. 1].

One property of interactive systems is that more effective software tools are needed to facilitate program debugging [Evans 66, p. 37]. Unfortunately, few tools provided any techniques not available on EDSAC (see, for example, [Schueler 77] and [Kazek 78]). In addition, "few high level languages provide facilities for interactive debugging in which the interaction is in terms of the high level language itself" [Hanson 78, p. 116].

3.4.1 Importance of interactive debugging.

Some have claimed that the programmer using an interactive system consumes more computer time and does a more superficial analysis of the problems than he would using a batch system [Model 79, p. 46]. In fact, however, other studies have shown that the overall elapsed time to find bugs is shorter by 50% to 300% with on-line systems and that the computer usage only 30% higher [Sackman 68]. A study by Boehm [71] suggested that enforcement of an interval between runs decreased the overall time required. In spite of this, there is ample evidence that users strongly prefer the interactive systems [Gold 69].

There are other advantages to on-line debugging. The system can give the user continuous guidance on the format of desired input, immediate feedback of errors, and control over output format [Naur 74, p. 252]. Also, Henriksen [77] notes, it is often difficult to locate a bug with the snapshot output available from batch runs. Only with interactive systems can monitoring be effectively used (section 1.3).

3.4.2 Examples of interactive debuggers.

The original DDT (originally for DEC Debugging Tape but more recently, Dynamic Debugging Technique) was developed at MIT for the PDP-1 [Kotok 61]. It allowed interrogation of machine registers, interpretive execution, breakpoints, single stepping, tracing, and patching of code. DDT-like debuggers have emerged for most assembly languages. Most allow use of the symbolic names of labels to specify locations.

Most debuggers for higher level languages have not expanded on the capabilities offered by DDT. Some, however, do attempt to allow the programmer to avoid having to know anything about the machine implementation of the language or the compiler. Examples of this are the PL/1 debugger on Multics, the IBM PL/1 checkout compiler [Satterthwaite 75, p. 23] and the current Mesa debugger (see section 4.2.1.2).

Some other debugging systems allow the user to correct mistakes using the source language and then continue execution. This implies that the full capabilities of the language should be available at debug time. An early system with this feature was the IMP system which had an integrated debugger and assembler [Lampson 65]. Most of the systems with this feature, however, are interpreted rather than compiled. Examples are APL, Basic, and MDL. Interlisp [Teitelman 78] is an interpretive system where ease of debugging was a significant design goal. Since Lisp is an interpreted language, it is easy for the system to allow the user to do arbitrary computations at any point. Interlisp also contains a powerful tracing facility that allows the user to investigate the call stack. A return can be enacted at any point with the return argument specified by the user. In

addition, any changes the user makes to the program, even the very procedure being run, are immediately reflected in the computation. The modified version is also saved [Teitelman 78, p. 15.3].

Interlisp also contains a system called DWIM (Do What I Mean) which attempts to correct errors as programs execute. If Interlisp cannot understand a string, it tries to find a spelling of that string that makes sense. If successful, the user is asked to confirm the new spelling or supply another one. DWIM also tries to correct unmatched parentheses and certain other common errors [Teitelman 78, pp. 17.1-17.28].

Unfortunately, interpreted languages tend to run slower than compiled languages [Satterthwaite 75, p. 24]. The ability to correct a mistake and continue is much harder to provide with compiled languages since "decisions made during compilation, such as those concerning the allocation of registers for temporary results, make the different pieces of the resulting machine level code inter-dependant" [Model 79, p. 53]. This makes *incremental compiling* (compiling only a small piece of code) very difficult. It is still used in some cases, however.

3.5 Dynamic and Pictorial Debugging Systems.

This section discusses a group of debugging systems that either use a CRT to allow monitoring of data, or produce pictures of the data, or both.

3.5.1 Non-pictorial monitoring systems.

The advent of CRTs as computer I/O devices allowed more effective monitoring of programs as they are executing [Gladwin 69]. RAID [Petit 69] provides the facilities of DDT, along with the ability to assign a variable to a particular place on the screen. The system will update the displayed value at breakpoints and while single-stepping allowing the user to monitor the program during execution. North [77] developed a system for an Intel 8080 microprocessor that updated the displayed values continuously by interpreting the code.

3.5.2 Pictorial static systems.

Yarwood [77] developed a system that would generate "illustrations" for programs in a very limited subset of PL/1. The pictures were generated on an electrostatic printer and were only useful for documentation of the program after it had been debugged. This system, however, is claimed to be one of the few attempts at analogical display [Model 79, p. 82]. In each snapshot, Yarwood's system displays a piece of text relevant to the state of the system along with a display of some data. A major feature is the ability to display one-dimensional arrays with the indices

displayed as pointers into the array. Parts of the array can also be labeled so that a level of abstraction can be shown (see Figure 3.1). A special language is used to specify what is to be displayed and when. This along with the original program is sent through a pre-processor which produces a new program which is then compiled and executed.

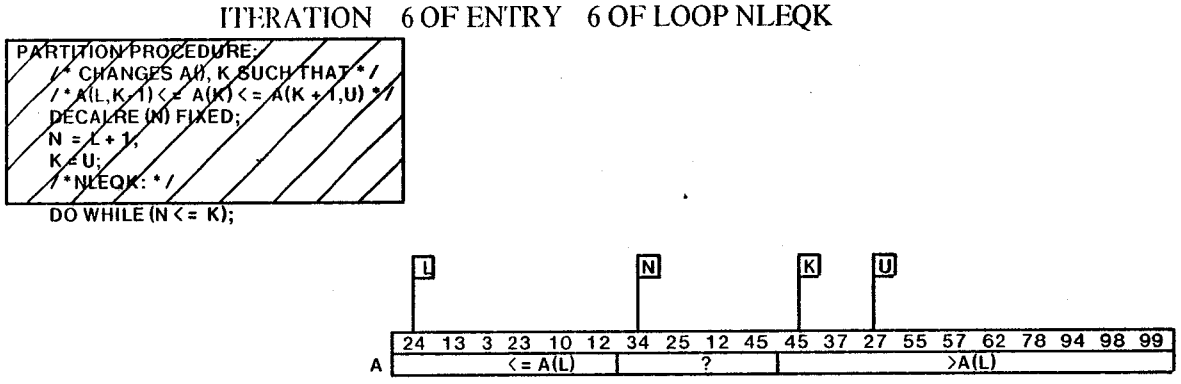


Figure 3.1 Yarwood's display of an array showing indices as pointers and labeled sections underneith. [Yarwood 77, p. 92]

3.5.3 Pictorial monitoring systems.

3.5.3.1 EXDAMS.

One of the earliest pictorial monitoring systems is EXDAMS (EXtendible Debugging and Monitoring System) [Balzer 69]. Unlike other systems, EXDAMS does not allow monitoring of running programs; the program under study must be run with an EXDAMS routine that collects information on a *history tape*. The information about the run can then be investigated at a later time. This history file allows the debugging aids to be language independent, since only the part that creates the history tape needs to know about the actual target program.

EXDAMS provides some very powerful display routines. For example, an inverted tree can be produced showing how a variable got a certain value (see Figure 3.2). The user can then request a similar "flowback" analysis along any of the resulting paths. Another static display available is a temporal list of all the values assigned to a variable. In addition, "movies" of the action of the program can be shown. The statement being executed will be highlighted and the values of displayed variables will be kept continuously updated. Also, as in Yarwood's system, the user can specify that a variable is an index into an array and have it displayed as an arrow. Balzer claims that the system is extensible so that even more interesting features could be added. In order for EXDAMS to work, it has to save a great deal of information. The statements added to the source program increase its length by approximately a factor of three.

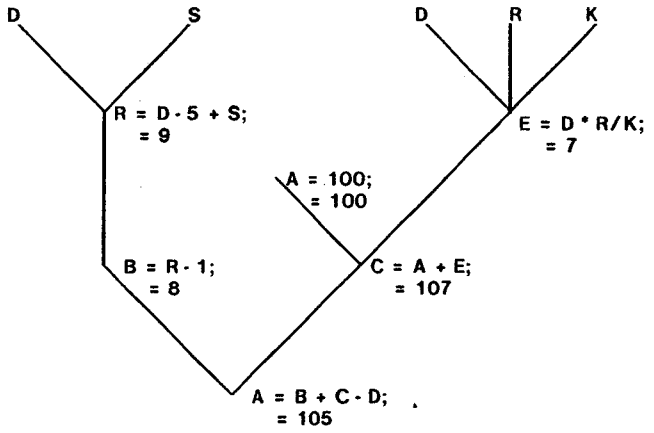


Figure 3.2. EXDAMS "flowback display showing how *A* got its current value. [Balzer 69, p. 569]

3.5.3.2 COPILOT.

COPILOT, a interactive programming system useful for debugging [Swinehart 74], was the first to exploit "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" [Teitelman 77, p. 1]. COPILOT is a dynamic system that "allows the user to create, modify, investigate and control programs written in an Algol-like language, which has been augmented with facilities for multiple processing" [Swinehart 74, p. xiv]. Central to the design is the use of multiple CRT displays that allow the different processes all to show their current state simultaneously. A major design goal was to allow the user to input commands at any time and not have to wait for completion of tasks. Thus "the user's terminal is continuously available for commands of any kind: program editing, variable inquiry, program control, etc" [p. xiv]. In addition, no process can prevent the user from directing input to another process. This is called the *non-preemption* property. Since the user's understanding of a display is dependent on the context, and there are many different contexts in a multiple-processing system, the environment in which a value was generated is displayed along with the values. Unfortunately, the processing power was not available to have the displays continuously kept up-to-date so the system operates using snapshots [p. 73], nor was the performance of COPILOT good enough to support actual users.

3.5.3.3 Smalltalk, windows and selections.

Smalltalk [Shoch 79][Ingalls 78] is a language system that incorporates a large number of display facilities. It was felt that graphics would make the system easier to learn and use [Kay 77]. Smalltalk developed the idea, first proposed in the FLEX system [Kay 69], of using multiple overlapping rectangular regions called *windows* to extend the available screen space [Goldberg 79]. The display for FLEX was a "large virtual screen on which displays may be 'tacked' like notices on

a bulletin board" [Kay 69, p. 235]. Windows can be moved in 3 dimensions. They can be translated to any portion of the screen, possibly changing size, and they can be moved "forward" (or "back") so they are less (or more) occluded by other windows (see Figure 3.3). The information in the windows may not be able to fit in the area specified, in which case only part of it is shown. The rest may be seen by *scrolling* the window the way one might move a scroll of text behind a small opening. The FLEX system incorporated the added feature of a true zoom where the displayed objects increase in size. This feature was not carried over into any of the later systems, however. Smalltalk presents a uniform window interface both to the programs and the user, thereby allowing complex systems to be easy to use (*e.g.*, an animation system [Backer 76] and Thinglab (section 3.6.4)).

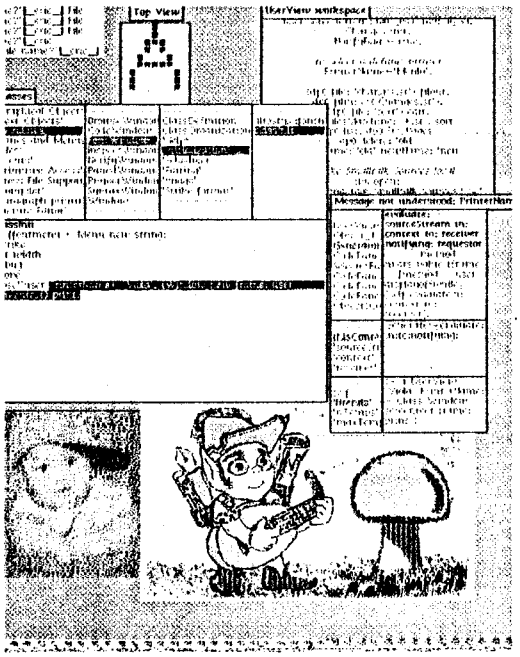


Figure 3.3 A typical Smalltalk screen showing multiple overlapping windows including a font-editing window (at top) and various types of pictures. [Picture courtesy Glenn Krasner].

Another important aspect of a display-based system such as Smalltalk is the use of *selection* to specify commands and their arguments. Selection using a *light-pen* to point at objects on the screen was used as early as 1963 for Sketchpad (section 3.6.2). Its applicability to interactive debugging has been long known [Zimmerman 67]. English [67] is credited with the first use of a *mouse* as a pointing device to select portions of the display (section 4.1.1). The advantages of using selection over type-in is that it is faster and less prone to errors. In addition, the user does not have to remember the corresponding text. Selections are closely tied to *menus*, which are lists of commands where the selected command is executed. Figure 3.4 shows two menus from the Smalltalk system.

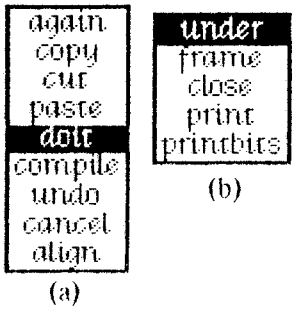


Figure 3.4. Two Smalltalk menus. (a) is the editing, compilation and execution menu and appears when the *Yellow* mouse button is hit. (b) is the window menu and appears when the *Blue* button is hit. [Picture courtesy Glenn Krasner].

3.5.3.4 DLISP.

Another programming system that facilitates debugging is DLISP (for Display Lisp) [Teitelman 77]. This system is built on top of INTERLISP (section 3.4.2) and thus has all of the debugging facilities of that system. In addition, DLISP uses multiple windows on a CRT to allow the user to interact with multiple processes using only one display. The windows are allowed to overlap (see Figure 3.5) and are treated essentially the same as Smalltalk windows. DLISP also contains primitives that make it easy to create pictures. For example, Bill Laaser was able in just two weeks to create a package that drew pictures of actual Lisp lists such as in Figure 3.6 [Laaser 79]. In another data-display application, Figure 3.7 shows a pretty-printed list output and a pictorial presentation of the same tree data structure. It is clear that the latter is much more evocative.

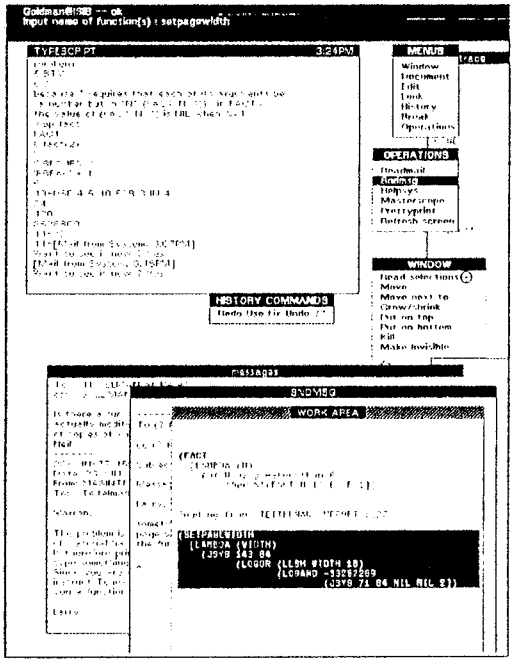


Figure 3.5 DLISP screen showing windows overlapping during a typical session. Text in *WORK AREA* on black background is selected. [Teitelman 77, p. 18].

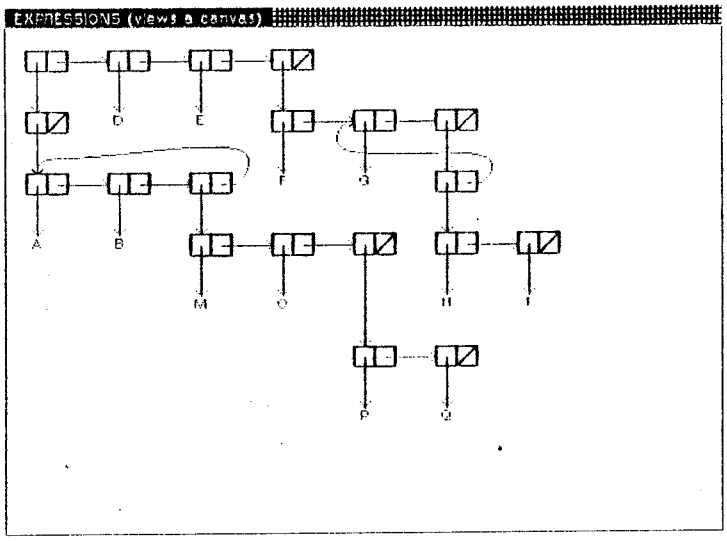
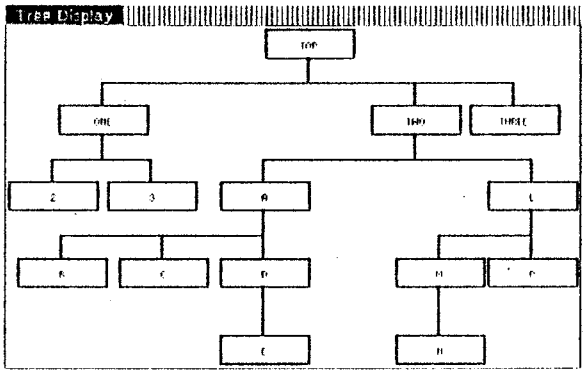


Figure 3.6 Pictorial display of the list: (((A B (M O (P Q))))
D E (F G ((H I))))
in DLISP after being modified by selecting cells and specifying where they should point with the mouse. [Picture courtesy Bill Laaser].



(a)

```
(TOP (ONE (2)
          (3))
     (TWO (A (B)
             (D (E)))
          (L (M (N))
             (P)))
     (THREE))
```

(b)

Figure 3.7 Pictorial display in DLISP (a) of the tree structure shown in (b). [Model 79, p. 115].

3.5.3.5 Sweet's tree drawing system.

Another system for making trees from internal data structures was done for Mesa by Sweet [78]. This was incidental to his major project but proved invaluable to the understanding and debugging of his program. His trees are produced on a fixed-width character output device (such as a line-printer) and have the following properties:

(1) All nodes at a given level in the tree are at the same level on the page;

(2) Each non-terminal node is centered over the nodes of its sons; and

(2) The width of the resulting tree is minimized [Sweet 78, p. 91].

A sample of the output of his system can be seen in Figure 3.8.

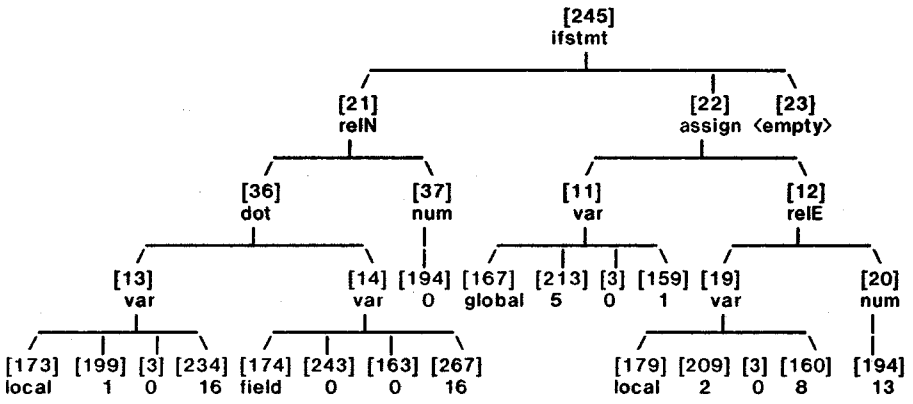


Figure 3.8. Sample of Sweet's Mesa tree display.
[Sweet 78, p. 89]

3.5.3.6 Model's system.

Model [79] attempted to blend some of the good ideas presented above into a system that would allow the debugging of complex artificial intelligence programs. His system was demonstrated monitoring KRL [Bobrow 77] and Mycin programs, but was designed to handle a more general class of programs. It is built on top of DLISP and uses many of its sophisticated display facilities. In Figure 3.9, a snapshot of a sample run of Mycin is shown. The Mycin program was actually running at Stanford while the monitoring program was running at PARC and communicating over the ArpaNet [Kahn 72]. Model's system keeps the different windows consistent by showing the progress and actions of the monitored system at all times. As in COPILOT, the multiple windows allow the information to be displayed in a highly organized manner. Thus, "many pieces of information can be presented in a constantly changing display, but the user need only look at those pieces which are of immediate interest" [Model 79, p. 47]. A major feature of the system was the ability, as in SNOBOL (section 3.1.4) to define events in programs and have monitoring activity take place when they occurred. In addition, a history was kept, allowing the investigation of any processing after a run "perhaps at different speeds, levels of detail, or foci of attention" [Model 79, p. 13]. All investigations are in terms of the Mycin or KRL language; the programmer never had to know about the underlying Lisp implementation of these languages, much less the machine implementations of LISP.

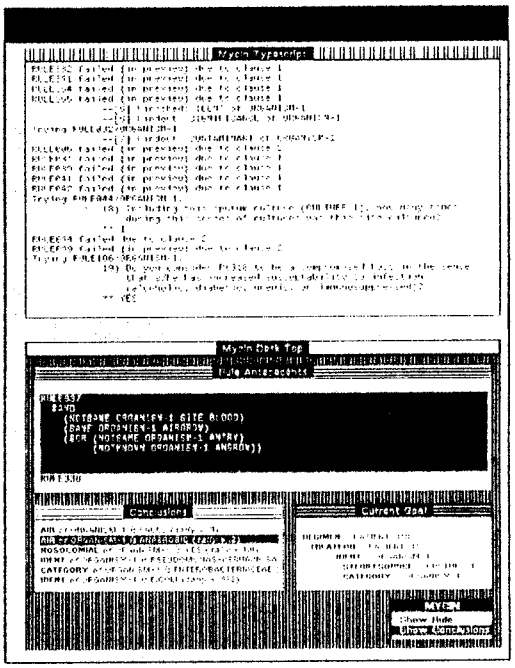


Figure 3.9 Example of Model's display for monitoring Mycin. [Model 79, p. 156].

Although Model discusses analogical display a great deal in his thesis, his display is not really analogical. As can be seen in Figure 3.9, the information is all presented in textual form. He does mention that a tree such as shown in Figure 3.7 could be made for the trigger structure of KRL [p. 145]. This was apparently not exploited, however. Another important problem with Model's system is that it requires the full power of a large computer to run and even then is very slow [Fikes 79]. This is partially due to the slowness of DLISP itself.

3.6 Graphical Systems.

The systems that have attempted to create dynamic, analogical displays have, for the most part, not been directed at debugging. These systems did develop many of the central ideas used in Incense and other graphical display systems, however. AMBIT/G, Sketchpad and Thinglab were designed as unified systems with graphics as a central part. The earliest systems to attempt to use computer generated graphics to help in the understanding of programs were the automatic flowchart generating programs (for example [Knuth 63], [Hain 65] and [Greene 73]). Automatic logic diagram layout systems (such as [Aaronson 61]) were another early application of computer generated graphics. The layout algorithms used in these systems were mostly trivial or excessively complex and had no influence on Incense.

3.6.1 Early analogical display.

Another example on early work at graphically displaying information by computer was motivated by a completely different problem. The Whirlwind computer, built at MIT in 1951, had only mechanical typewriters for character output, and these were very slow. Therefore, an oscilloscope was attached to the computer along with a computer controlled camera. Programmers were encouraged to have the computer make graphs of their data rather than trying to type it out [Bell 71, p. 139]. Soon, however, line printers were developed and having the computer generate the pictures became more expensive than making listings.

3.6.2 Sketchpad.

Sketchpad [Sutherland 63] was an early system that attempted to simplify the process of using the computer to make pictures. It used a light-pen, four knobs, and a tremendous number of toggle switches to allow the user to input information analogically (for the most part) [p. 25]. The basic building blocks available were line segments, circle arcs, and text. The system also allowed the user to define *symbols* out of a set of these which then could be used in higher level objects. If the definition of a symbol was changed, all instances immediately re-shaped themselves. Instances could be different from the original (prototype) by having different location, size (scaling), and rotation [p. 46]. Symbols had the additional property of *connection points*, which were the only places at which additional objects could be attached. The system also incorporated some powerful mathematical constraints that allowed the user to specify relations that had to be preserved among various objects displayed. Another important feature of the system was that "the organization of Sketchpad display as a set of display routines with identical external properties [made] it possible to add new kinds of displays to the system with the greatest of ease" [p. 42]. Drawings could also be saved in a manner that allowed "cartoon motion pictures" to be displayed [p. 67]. Sketchpad was a powerful system and it had a major influence on almost all subsequent graphics systems.

3.6.3 AMBIT/G.

One such system is AMBIT/G (Algebraic Manipulation by Identity Transformation/Graphical) [Christensen 67]. This system attempted to allow the user to specify his program entirely with pictures through the use of a pattern matching language. The results of the computation were then presented pictorially. An AMBIT/G program is composed of a generic set of *shapes*, instances of those shapes, a *data graph* connecting the instances, a set of *program statements* and a *control structure* connecting the statements [Henderson 69]. The programmer "usually endows each shape with some distinct interpretation" [Henderson 69, p. 1]. A property of the shapes is the fixed number of *links* that are allowed to leave the node; a node can have an arbitrary number of links *to* it, however. Links were all drawn as straight lines with arrowheads in the system implemented.

The layout of the data graph in the AMBIT/G language is irrelevant to its meaning and there was no automatic formatting. In fact,

> the problem of automatically laying out and displaying an entire data graph has been carefully avoided; the user is required to specify small parts of the data graph that he wishes to see, and he is encouraged to aid in the layout of these [Rovner 69, p. 12].

To display a data graph, the user specified a uniquely named node and where it was to go on the screen. If he then wished to see sub-nodes, he had to point at each connection point he wanted to see expanded. The sub-node would then be drawn at a canonical place relative to the parent irrespective of what may have been there previously [Rovner 79]. If the subnode was nil, an "*" (asterisk) would be drawn, and if the subnode happened to already be displayed, the link would have been drawn to the original instead of to a copy. If a node was drawn in an inconvenient place, the user could move it and the links would redraw themselves correctly [Rovner 69, p. 11].

AMBIT/G has a number of techniques for recognizing user commands. The programmer used a tablet and pen with which he could draw and select objects. The system also recognized a number of *gestures* such as the "scratching out" of a line to specify deletion. In addition, there were displayed menus to allow selection of some commands [Rovner 69, p. 10].

AMBIT/G, unfortunately, was slow (typical delay was 10 to 20 seconds [Rovner 69, p. 13]) and "so general that it is difficult to build and use, and it has never been completely implemented" [Christensen 71a]. Efforts were therefore directed at specifying related but more practical languages. AMBIT/L [Christensen 71a] has a limited and fixed number of predefined shapes which support list structures. IAM, a system for interactive algebraic manipulation, was implemented using it [Christensen 71b]. Proposals for future research included the ability to monitor the system while manipulating the data graphs by having the modifications shown as they occur [Rovner 69, p. 13].

3.6.4 Thinglab.

Thinglab [Borning 79] was written in Smalltalk to study aspects of a constraint oriented system. It was based in a language which had powerful facilities for interacting with the screen (section 3.5.3.3). The system allows multiple views of an object to be visible at the same time and "a typical object can be depicted in several ways.... The object itself defines the views that it can provide" [p. 4]. For example, the user can specify a constraint that the height of a bar in a bar graph corresponds to the value of some integer in a text paragraph. When one is changed, the system forces the other to re-adjust itself so they are once again consistent [p. 4]. The collection of all the constraints on an object may be "incomplete, circular, or contradictory" yet the system manages to sort this out [p. 5]. When the user specifies that he wishes to display an object, a menu of all its possible ways of displaying it is presented and the user can pick one. He then specifies where the object is to be placed [p. 15]. As in Sketchpad, to which this system owes much, objects all use the

same protocols making the generation of new classes easier. In fact, some classes can be specified simply by drawing a prototypical example (*e.g.*, making a class *trapezoid* from the class *quadrilateral*). For a given class, the prototype "is a distinguished instance that owns default or typical parts" [p. 46] which are inherited by instances unless overruled. This prototype may be NIL except for graphical objects which must have some specified appearance.

The Thinglab system is claimed to have good response time for objects with simple (linear) constraints. The time to arrive at consistent values is "usually as good as if a suitable method had been hand-coded" [p. 51]. If constraints are circular, however, the system must use a relaxation technique which is much slower.

IV. The PARC Environment for Incense

Incense incorporates some of the good ideas from earlier systems. In addition, Incense reflects many of the constraints and capabilities provided by the environment in which it was designed and implemented. The Palo Alto Research Center (PARC) has extensive facilities available to aid in the development of systems such as Incense. In hardware, there is the *Alto* mini-computer and two types of faster research machines. In addition there is a large body of software written in the PARC language Mesa that was useful in building Incense. The Computer Science Laboratory (CSL) at PARC is currently in the process of developing a new environment for Mesa called *Cedar*, and it will have additional facilities that will be useful for future versions of Incense. This chapter briefly discusses each of these facilities so that the Incense system can be understood in context.

4.1 Hardware.

The *Alto* [Thacker 79] is a general purpose, microprogrammable mini-computer designed in 1973. The standard configuration of the Alto includes (see Figure 4.1):

- An 875 line raster-scanned display;
- A keyboard, a "mouse" pointing device with three buttons, and a five-finger keyset;
- One or two 2.5 Mbyte removable cartridge disks;
- An interface to the Ethernet distributed packet-switching local computer network ("Ethernet"), a 3 Mbit/second communications facility [Metcalfe 76];
- A microprogrammed processor that executes programs, controls input-output devices, and supports up to 3K of user-programmable micro store RAM (along with the 1K of PROM); and
- 64K 16-bit words of semiconductor memory, expandable to 256K words [Thacker 79, p. 1].

Figure 4.1. Picture of typical Alto work-station with keyset, screen, keyboard, mouse, computer with two disk drives.

Incense does not currently use the keyset or Ethernet, and it requires an Alto with extra micro-programmable RAM (for loading of special micro-code for doing real number arithmetic) and at least 128K of main memory to hold all of the data and programs. The next sections describe the most important aspects of the Alto for Incense: the mouse and the screen.

4.1.1 The mouse.

The mouse (see Figure 4.2) is a pointing device which fits comfortably under the hand and can be rolled around on any frictional surface [English 67]. The mouse buttons (which are called *Red*, *Yellow*, and *Blue*) allow the user to specify a number of actions using the same hand with which he is pointing. The current state of these buttons (up or down), along with the mouse position, are available through high-level abstractions in Mesa.

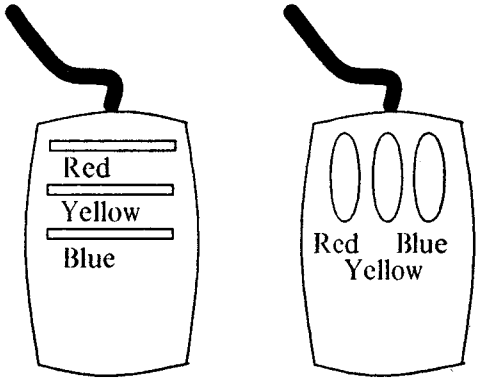


Figure 4.2. Two styles of mice with buttons labeled.

Since the mouse only measures relative movements and not an absolute position, it is essential to have visual feedback as to where the mouse is with respect to objects on the screen. This is provided by the *cursor* which follows the mouse. The actual picture shown at the cursor location can be set by the user. The default is a simple left-pointing arrow (Figure 4.3a). Even though the area of the cursor is small (about 1/4" square) a large amount of information can be presented in it. For example, the system Okra, built by the author to interface the Alto to a remote file server, utilizes 15 different cursors which clearly show the state of the system (Figure 4.3). Incense currently does not take advantage of this capability, however.

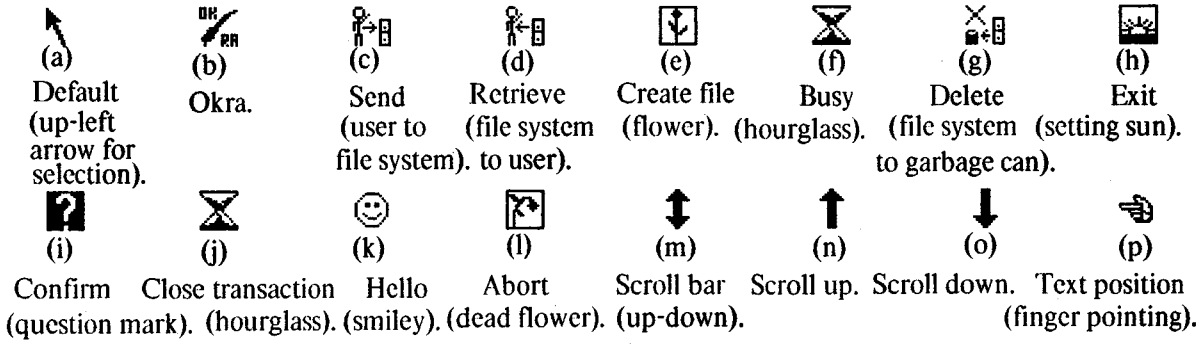


Figure 4.3. Cursors used in the Okra system. Incense uses only (a).

4.1.2 The screen.

The Alto display is an interlaced 875 line monitor running at 30 frames/second. There are 808 visible scan lines, and 608 picture elements (*pixels*) per line. It is oriented with the long dimension vertical, and the screen area is about 8½ by 11 inches (Figure 4.1). The actual picture on the Alto screen is controlled directly by the contents of a set of *bitmaps* or frame buffers that are stored in memory. Each bit of the bitmap corresponds to one pixel on the screen and determines whether it is on or off. The Alto screen has the capability to use multiple bitmaps for the display, but Incense does not use this feature. The standard microcode provides one very useful function called *BitBlt* which can transfer an arbitrary rectangle from one place to another in memory. Since the picture presented is stored in memory, BitBlt allows arbitrary rectangles on the screen to be moved. The resulting display can be a function of the source and destination rectangles such as XOR, OR, AND, etc. BitBlt is useful for filling areas, drawing lines and displaying text.

Since the bitmaps are stored in main memory, it is possible to trade off the size of the picture and the amount of memory available for other data. For example, a full screen requires

(808 lines/screen * 606 bits/line) /16 bits/word = 30603 words

or nearly half of the 65K allowed for data in an Alto. Therefore, a full screen is seldom used in applications such as Incense where there is a lot of other data.

4.2 Software.

There has been a great deal of software written to help the programmer at PARC. Incense is written in Mesa so it takes advantage of the facilities of the Mesa system. In addition, two special systems were used in the development of Incense. CGraphics, written by John Warnock at PARC, provides the basic interface to the screen, and JAM, designed by Warnock along with Martin Newell, is an interpretive environment that was used to debug Incense.

4.2.1 Mesa.

Mesa [Mitchell 79b] is a large and complex strongly typed language, and there are a large number of programs and systems built in and around Mesa to aid in the production of software.

4.2.1.1 Compiler and symbol tables.

The current compiler for Mesa is batch oriented and operates on one file at a time, producing error messages in another file. The compiler operates fairly quickly and produces efficient code. In addition to its main duty of producing the object code, the compiler also produces very complete symbol tables. These are used to allow separate compilation of different modules and allow Mesa-level debugging of programs. The symbol tables contain sufficient information to discover the location in memory and type of *all* the variables and constants used in the program, to find the source statement corresponding to a value of the program counter and vice versa, and to resolve references to types declared in different modules. As a result, symbol tables tend to be very large, typically taking up about four times as much disk space as the object code for the program. Symbol tables and Mesa types will be discussed further in chapter 6.

4.2.1.2 Current Mesa debugger.

Mesa, unlike many high level languages, has a very powerful debugger that allows investigation of a program almost entirely in terms of the abstractions of Mesa. There is no requirement, or even advantage (in most cases), to know anything about the machine instructions or data formats. The extensive symbol tables provided by the compiler allow the debugger to know about user-defined types, local variables, enumerated types, and aggregate data structures. It also allows the setting of breakpoints by pointing with the mouse at a source text line and executing the command *set-break* (see Figure 4.4). The debugger has a limited interpreter for evaluating expressions and some statements. The source program and its execution speed is not affected by the presence or use of the debugger. Another advantage of the current debugger is that it is uses a screen package. This allows the user to have multiple windows on different files. Snapshots of actual debugging sessions are shown in Figures 4.4 through 4.7.

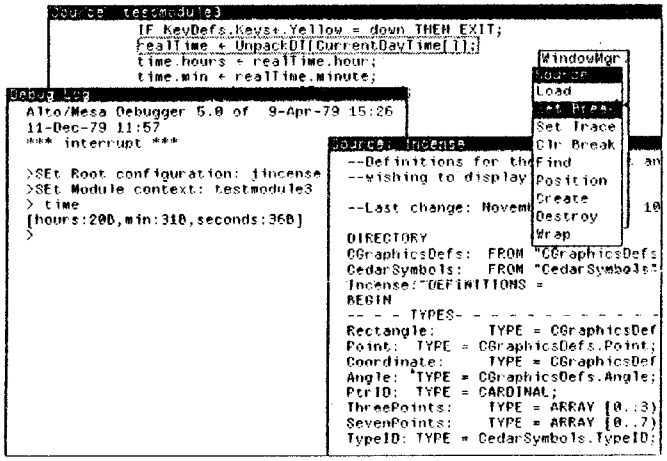


Figure 4.4 Mesa Debugger screen showing debugger window, two text windows, and a menu of source commands with *Set Break* selected. The break will be set before the statement selected in the TestModule3 source window:

realTime ← UnpackDT[CurrentDayTime[]];

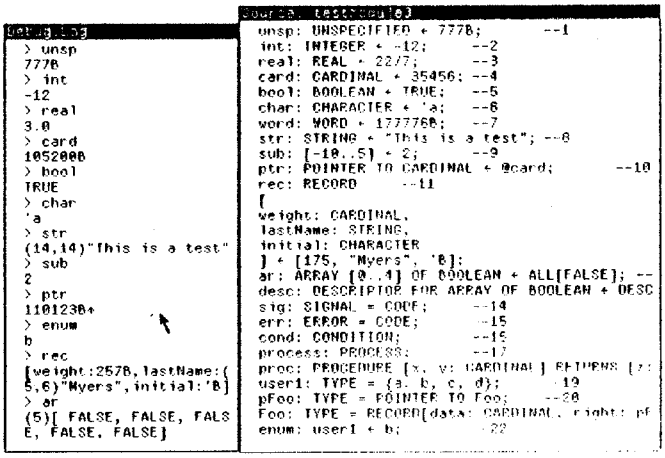


Figure 4.5 Mesa Debugger screen showing display of some variables from module "Testmodule3". Note format of cardinal, pointer, record, and array display.

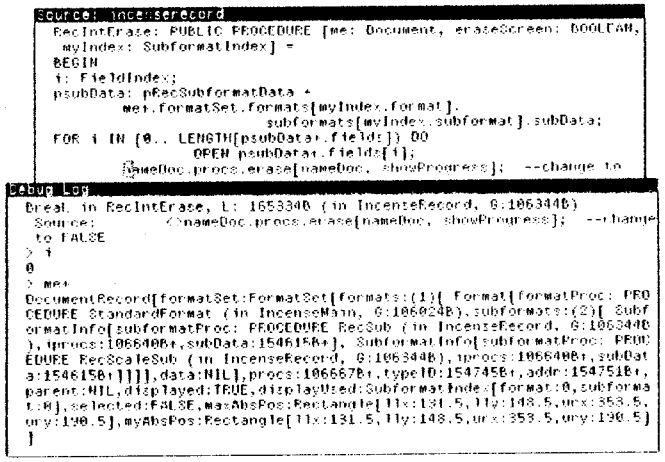


Figure 4.6 Mesa Debugger screen showing display of a large record (a *Document;* see section 5.3) containing arrays and pointers.

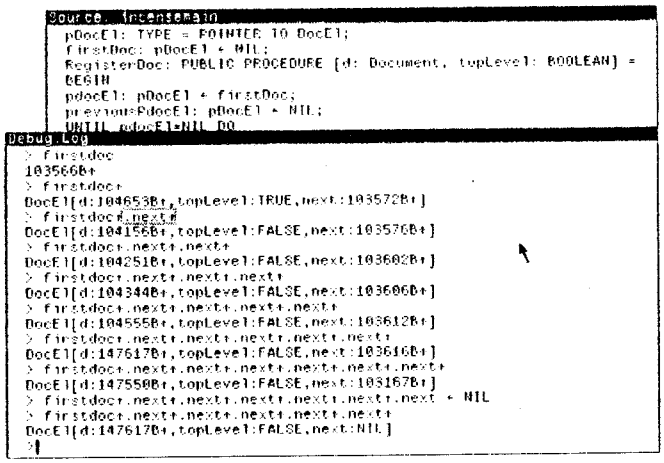


Figure 4.7 Mesa Debugger screen showing an attempt to investigate a pointer structure. Note the assignment shortening the list by making a next field NIL.

The debugger does have some limitations, however, which irritate a number of people. The main complaint is that it runs too slowly. The user can invoke the debugger while a program is running or by setting breakpoints. In either case it takes about two seconds for the debugger to be installed. Similarly, when continuing from a break, it takes another two seconds. This makes it very tedious to single step through a program or to monitor variables. "The real problem," according to one Mesa user, "is that I can think a lot faster than the debugger." There also is no control over the way variables print, so, for example, the user cannot request that only certain fields of a record be displayed.

Another limitation of the debugger has to do with multiple processes since inspecting their various states is usually awkward. Some people also complain about the inability to display lists,

trees, hash tables, and other sparse or pointer-based data structures conveniently (see Figure 4.7). In addition, the interpreter is very limited. It incorrectly handles certain Mesa types and will not allow any memory allocation operations such as creating new strings or temporary variables. [The information in this section is from personal experience and the results of a debugger poll described in Appendix A].

4.2.2 CGraphics: The underlying graphics package.

CGraphics is a graphics package written by John Warnock at PARC that allows the client to use the abstractions of *lines*, *areas*, and *text* rather than the low level BitBlt operations. In addition, the same commands can be used to draw on an Alto screen, a black and white or color TV display, or to hardcopy output files. Higher level systems such as Incense can therefore be independent of the display type.

A basic data structure in CGraphics is a *Display Context* (or *DC*). Every drawing command uses a display context to determine how the command will operate. The display contexts contain information about the current position, scaling, rotation, clipping boundary, text font, font size and style, area and line colors and textures, and painting functions. The DC used is either an explicit parameter to the functions, or the top of a stack of DCs which is maintained by the system.

Some of the drawing functions provided by CGraphics are: draw a line at any angle, draw a rectangle outline, fill a rectangular area, draw an arbitrary polygon or fill its interior, and put up text. It is also possible to draw a parametric cubic polynomial curved line [Newman 79] (in particular, a *spline* [Ahlberg 67]) that goes through a specified set of points called *knots*. Finally, there are routines for modifying a display context, for finding out the size of a string, and for transforming a rectangle or point from the coordinate system of DC to that of another. In the future, there will also be routines for filling an area defined by splines and for finding the intersections of regions. Figure 4.8 shows an example of the current capabilities of CGraphics.

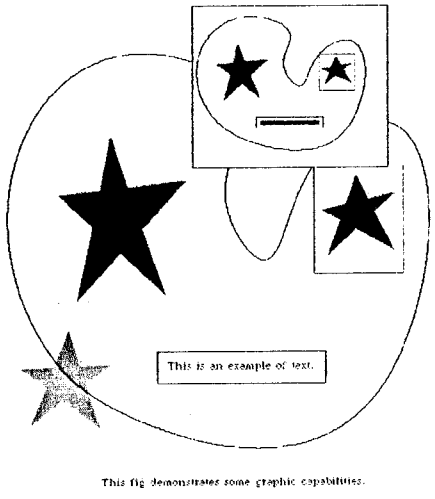


Figure 4.8. Demonstration of some capabilities of CGraphics.

4.2.3 JAM: An interpretive environment.

JAM is an interpretive system written in Mesa. It has its own syntax and command language that is strictly reverse-polish. Incense was debugged using JAM as the user-interface and currently only exists in the JAM environment. If Incense were to be released, it would have to be removed from JAM (a simple and straightforward task) and an alternative method of specifying the Incense commands would have to be implemented. This was not done for the current Incense since it seemed more appropriate to await the Cedar paradigms.

4.3 Cedar: A Future Environment for Incense.

Incense is actually a prototype for a component of *Cedar*, a future programming environment of the Computer Science Laboratory (CSL) at PARC. (In the world of botany, Incense is a type of Cedar: *Calocedrus decurrens*.) Many of the facilities designed and planned for Cedar have a direct impact on Incense. Incense was designed to utilize all these features, but many were not available at the time of its implementation. Section 8.2 will discuss how Incense might be modified to use these additional features.

One of the most important changes to Mesa envisioned for Cedar is a garbage collector. This will free the user from having to worry about storage management. Adding this feature to Mesa required that a subset of current language be defined, called the *safe language*, that eliminates any possibility that the programmer might destroy the garbage collector's state. The safe language contains restrictions on the use of pointers and on breaches of the type system.

One of the Cedar committees is the *User Facilities Group*, under which the Incense project was conducted. This group was given the task of defining how the users of Cedar would interact with the system and his programs. It was decided early that many of the features of DLISP and Smalltalk were required in Cedar, such as a history facility to allow replays, and a uniform manner of accepting keyboard and mouse inputs from the user. In addition, an abstraction called a *document* was defined to allow operations on the screen. Documents play a central role in Incense and are described in section 5.3.

V. Incense – An Overview

In chapter 3's survey of debugging and graphical systems, there were no systems presented that could claim to provide analogical display of data in a manner that would allow truly interactive debugging. In chapter 2, however, it was posited that this is an important feature for a debugging system to have. Incense attempts to fill this void, but since it is actually a *prototype* rather than a production system, many desired features are not included. Incense does manage, however, to demonstrate many of the advantages of automatically generated analogical displays for actual program data structures.

This chapter describes the goals of Incense as motivated by the discussion of chapters 2 and 3, and then presents an overview of the design. The next chapter describes the design and implementation of a run-time type system called *CedarSymbols*. It was created by the author to allow Incense to discover the types of actual Mesa variables. The chapter following presents some details of the current implementation of Incense and its performance and limitations.

5.1 General Goals for Incense.

This section summarizes the important goals of debugging and data display systems discussed in chapters 2 and 3, and mentions how they relate to the actual design of Incense.

Easy to use. If a debugging or data display system is to be popular or even tolerated by programmers, it must present a natural and pleasing user interface. Many systems require the programmer to put special commands into his code, to re-compile, or to specify the desired display in a special language (as in Yarwood's system: section 3.5.2). Incense was designed to have reasonable and understandable displays for data structures. The interactions required to select and modify the display or the underlying data are similarly straightforward and natural.

Extensible. One major problem with the current Mesa debugger is that there is no way for the user to modify the way that information is presented. It is also difficult to fix the debugger when there are changes to the Mesa language. Many of the systems discussed (*e.g.*,Yarwood's, section 3.5.2), allow the user to control the display but require the use of special languages. Incense was designed to allow the programmer maximum flexibility in designing the displays: He is allowed to specify where data is to be displayed; he can choose among a set of predefined displays for any particular data structure; he can modify that display in certain ways; and he can construct actual programs (in Mesa) to define the display. In addition, the programmer can associate a certain display style with a variable or type so it will be used whenever the variable or instances of the type are displayed. Finally, the programmer should be able to use any of these capabilities during a debugging session.

Analogical. Section 2.2.4 argued that displays should be easy to read and pleasing to the eye. Few debugging systems fulfill this principle (see, for example, Figure 4.7). A pictorial display for data structures would make the structure of the data much easier to understand. This might then make the debugging task swifter and more enjoyable. Incense had as a crucial goal the capability for analogical display. To achieve this, Incense uses the graphical capabilities inherent in the Alto to provide displays at as high a conceptual level as possible.

Fast. The most common complaint about most systems is that they run too slowly. The Mesa debugger suffers from this problem. People can become accustomed to even the most complex interface, but they tend to be continually frustrated at delays. DLISP (section 3.5.3.4) is a very exciting and powerful system, but it is seldom used because the response time is so long [Teitelman 79]. Incense is a prototype system (as is DLISP), so speed was *not* a design goal. Incense does, however, display most data at an acceptable rate, even faster than the Mesa debugger for certain cases. Running on the faster of PARC's new research computers, Incense should produce displays very quickly.

5.2 The Incense System.

Incense is a working system that displays data structures analogically. All of the illustrations in this chapter and the two following were created by Incense and taken directly from the screen. Defaults displays are generated automatically based on the type of the data. These display formats are general enough to handle almost all data structures in a reasonable manner. In addition, the programmer is given the option of specifying and modifying the displays at various levels. The next sections discuss how this was achieved.

The most difficult aspects of Incense were (1) allowing the client to define new display formats, and (2) the display of pointers. The solution to the first problem involved using a carefully designed abstraction to hide the internal data and procedures (see section 5.3). The problem with pointers is that a location on the display must be chosen for the referent. *Layouts* were invented to handle this problem (see section 5.3.1.3).

5.3 Documents: The Basic Component of Incense.

In order to have some data displayed in Incense, a *document* must be associated with it. The term arose in Cedar where *documents* will be the entities that can display themselves on the screen. A major item that will be displayed by Cedar programs is text, in particular, computer programs, memos, and business letters. These are generally grouped under the heading *document*, so Cedar used the concept as a metaphor for all displayable entities.

Chiefly to allow Incense to be extensible and uniform, an *object-oriented* or *data-abstraction* approach was taken. Documents are therefore organized as a data objects: each instance keeps internally all of its state and all of the procedures that are allowed to modify the state. Thus, documents are like instances of CLU data abstractions [Liskov 77] or Smalltalk classes. All documents have the same basic structure including the types of their data and procedures. Thus the interface to all documents is the same. The actual procedures used in a specific document will be different for documents of different types, however. Thus the type of the document is defined by the actions of its procedures. For example, a document for an INTEGER would differ from that for a BOOLEAN in that they would have different procedures for interpreting the value in memory and translating that value into a textual representation that could be displayed.

The association between the data structures and the documents to display them is automatically created by the Incense. The memory address and type of the associated data structure are stored in the document along with the appropriate procedures and other data. There are six basic classes of procedures required for documents in the current Incense system: procedures for display, erasure, selection, editing, de-allocation, and drawing of arrows.

5.3.1 Displaying a document.

5.3.1.1 Formats and subformats.

A document for viewing a data structure would obviously not be very useful if it were not able to display itself. This is the most important operation of the document and also the most complex. Each document can display itself in an arbitrary number of ways called *formats*. A document must have at least one format, however. The formats are intended to be radically different ways of displaying the data. For example, a document for a certain type of record (Figure 5.1) might contain formats for displaying itself as a normal record (a) or a clock (b). The client is required to specify which format should be used. Currently, all automatically generated documents contain only one format.

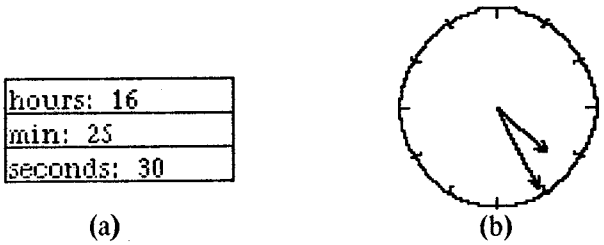


Figure 5.1. Two formats for the record document for:
Time: RECORD [hours, min, seconds: CARDINAL];
holding the time of day: (a) as a normal record and (b) as an analog clock.

Each format contains an arbitrary number (nonzero) of *subformats*. The subformats specify the actual display and are chosen automatically by the system based on various contextual information, such as the size of the area in which the document is to be displayed. Thus, an array document (Figure 5.2) might contain two subformats. The first would display the data normally (a). The other subformat would be used only if there was not enough room, and would use grey rectangles for the values (b). The client cannot choose a subformat explicitly. Instead, the creator of the document associates a test with each subformat to determine whether it is applicable in the current context. Since more than one of these tests may succeed, the designer also specifies an ordering of the subformats. The *formatProc* procedure associated with the client-specified format will cycle through the subformats in order until one is found that can be used. If there are none, then the document will not be displayed. This may happen, for example, if the area in which the document is to be displayed is very tiny. The designer can force a document to be displayed by assuring that the final *subformatProc* will always return TRUE.

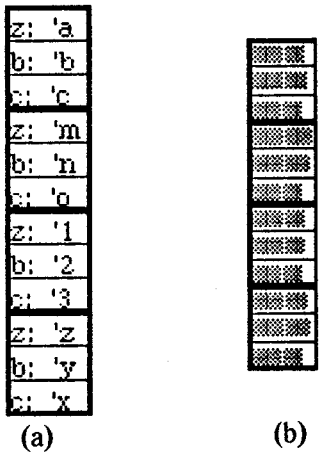


Figure 5.2. Two subformats for an array of records:
ar1: ARRAY [1..4] OF RECORD [z, b, c: CHARACTER];
(a) normal and (b) as grey areas due to lack of room in the Y direction. Note that the length of the grey areas varies depending on the length of the actual value.

The arguments to the formatProc are the document itself, the format chosen by the client, and a rectangle, called *maxArea*, into which the display must fit. Thus the invoking procedure (or the user) always specifies the size of the display and the document must be prepared to fit itself into any size rectangle. In most other systems, the display area specification is handled differently. For example, in AMBIT/G, the displayed objects always take the same amount of space, and in Smalltalk, the objects take as much room as they desire. In Incense, the user always has control of the placement and size of the displays. This feature also allows aggregate structures such as records to accurately specify the position and size of subparts.

The subformat procedures for some documents, such as those for RECORDs and POINTERs, can cause the display of subordinate documents. For example, the standard record subformatProc will iterate through the documents corresponding to each field, calling the formatProc in each. The maxArea rectangle for the subordinate will specify where the field should be placed relative to the rectangle for the record. This uniform structure hides all details of the type of the subordinates. Thus the record does not have to know, for example, whether a subordinate is an integer, pointer or another record (see Figure 5.3).

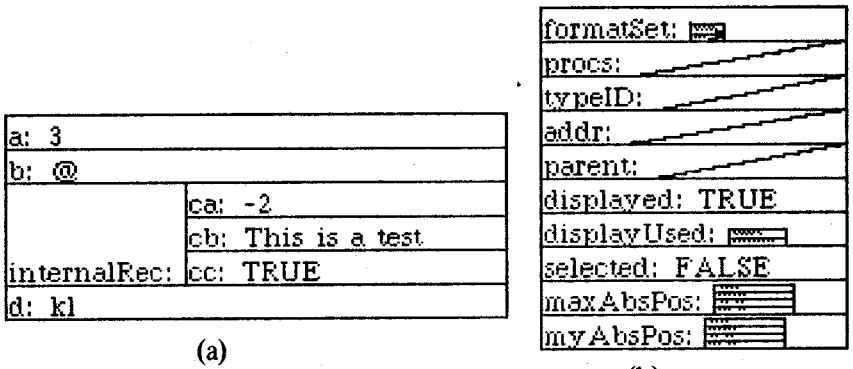


Figure 5.3. Two ways records can contain other records. Full size (a) and reduced (b).

Figure 5.4 shows the default display for all of the basic types. Note that the format chosen draws a box around the datum of exactly the correct size.

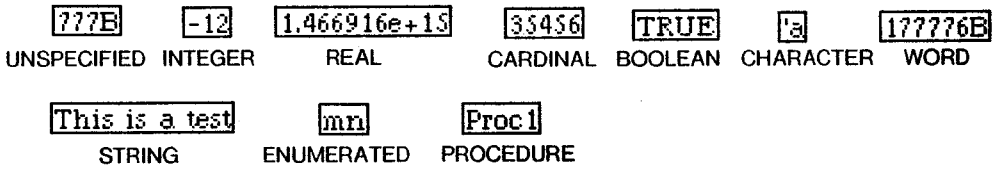


Figure 5.4. Default boxed display for the basic types.

5.3.1.2 The form and display data.

For aggregate data structures such as arrays and records, the document must know where to put the subordinates which will all be fit inside the rectangle for the aggregate. The relative locations are specified by some auxiliary internal data called the *form*. Each subformat has its own internal data, so that documents could conceivably rearrange their display automatically based on some criteria. In the current system, however, all the subformats for a particular format share the

same form data. For records, this data is a set of rectangles that specifies the field locations, the documents to be used to display the field, and a specification of the format to be used for each field document.

In addition to the form, which is constant throughout the lifetime of the document, other data is needed for a document that is on the screen. This data is called the *display data* and includes such things as the current screen position and the screen position where arrows should be drawn. Pointers also have the documents for the referent in their display data rather than in the form. If the value of the pointer was modified, the subordinate document would change and thus not be constant as the form is required to be.

5.3.1.3 Layouts.

The location of the subordinates for aggregates (records and arrays) is fixed relative to the aggregate's rectangle and easy to compute no matter how and at what level of nesting the record is displayed. With pointers, however, that is not the case. All documents must fit inside the rectangle provided for them by their caller. The rectangle for a pointer, however, specifies only the box for the pointer source end point. Therefore, the object pointed to must be put somewhere else. *Layouts* are a means for specifying where the documents corresponding to the destination of arrows should be placed. A layout has a field for the pointer or pointer-containing document, and one field for each object pointed to. Thus, for a record containing 2 pointers, a layout with 3 fields would be used: one for the record and one for each of the two referents (see Figure 5.5). Layouts and layout fields both have special documents that have no associated data or type, but simply serve to locate the various pieces.

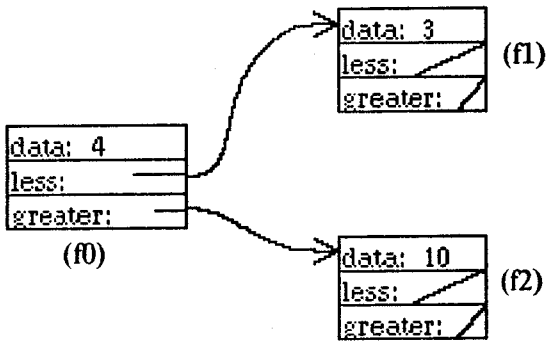


Figure 5.5. Layout with 3 fields: one for record: (f0), and one for each referent: (f1) and (f2).

An alternative strategy to layouts would be to have some global procedure allocate space on the screen for the pointer's referent. This would require a large amount of added complexity to locate, allocate, free, and compact rectangles of screen space based on various heuristics and constraints. Also, with automatic allocation, it would be difficult for the client to specify the space to be used if he wished to. All of the systems studied in chapter 3 (*e.g.*, AMBIT/G, Sketchpad, and Smalltalk) avoided this two-dimensional space allocation problem.

Currently, layouts use a very simple scheme for making the subcomponents fit into the area specified for the layout document. As with records and arrays, the rectangles for the various fields (the *form*) are fully specified at document creation time. When the layout is displayed, the subcomponents are simply told to fit into the specified area. Whenever a particular layout is displayed, the fields are placed in the same relative positions.

In a recursive structure such as Figure 5.6, there are layouts at each level of nesting. They get progressively smaller since the area provided for them is reduced at each level. This theoretically would allow the display of an arbitrary number of levels, but, in fact, after a threshold is reached and the documents are too small to see, displaying terminates. Pointers to documents that are already displayed do not cause an infinite cycle, since an arrow is simply drawn to the original occurrence (see Figure 5.7).

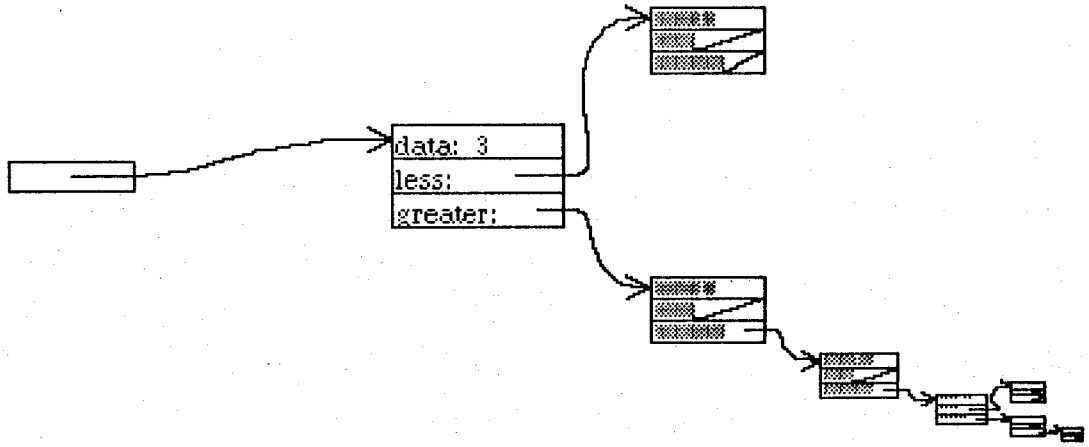


Figure 5.6. Deep recursive tree display demonstrating how elements get smaller. Overall structure, however, is easily understood.

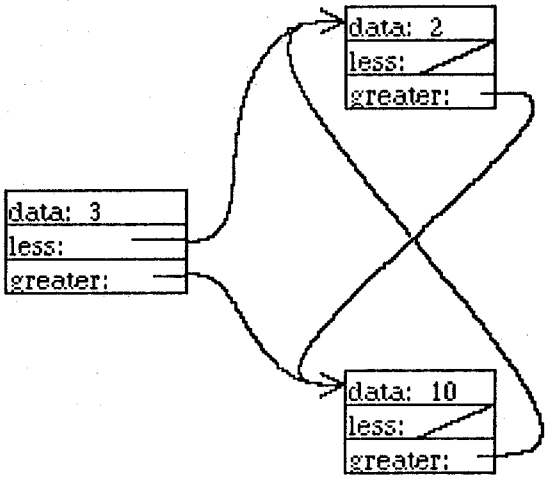


Figure 5.7. Pointer to previously displayed object does not generate a new copy. The second arrow is drawn to the first occurrence.

5.3.1.4 Prototypes.

One important feature of documents is that a client can design new ones and then associate these documents with particular variables or data types. Thus, the clock document could be associated with all data of type Time and then a variable such as CurrentTime below would be displayed as a clock:

```
Time: TYPE = RECORD [hours, minutes, seconds: CARDINAL];
CurrentTime: Time;
```

These *prototype* documents will never be displayed. The important information in them is copied into the documents for specific instances. Thus prototype documents have form but no display data.

Model [79, p. 74] says that "at the very least, system builders should provide formatted printing facilities for the system data structures they implement." Thus any major data type should have a prototype document built for it. A poll of current Mesa users (Appendix A) shows that many people would be willing to create documents for their data structures if it were simple and straightforward. In fact, many people do something similar currently: "I've always been in the habit of writing pretty-print routines for my more complicated data structures." Documents provide a structured way of doing this and promote the use of analogical output.

There are presently three ways of producing prototypes for a document. The most straightforward is to use the system default. This requires no knowledge or special actions. The second way is to write a Mesa program to define the display. The latter approach is clearly necessary for the more esoteric and unusual displays such as the clock. Finally, the client can specify the *form* for the document. For example, a user might draw boxes on the screen using the mouse to specify the relative size and position of the various fields of a record or array. Fields can be omitted simply by specifying a rectangle of zero size. Figure 5.8 shows an example of record form design.

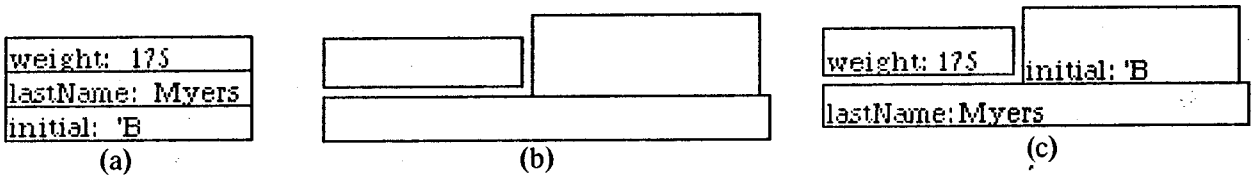


Figure 5.8. Normal display for record (a), form defined by user (b), and resulting display (c). Note that field order has been switched.

5.3.2 Drawing arrows.

Documents require three different procedures for properly handling layouts and pointers. The precise operation of these procedures will be discussed in section 7.2.3.2, but a general overview of the procedure that actually causes the arrow to be drawn will be given here. The document for the pointer calls the *DrawArrowFrom* procedure in the document associated with the referent. An argument of this procedure is the source location on the screen for the arrow. The target document must already be displayed for this operation to be successful since it needs to calculate the destination point of the arrow on the screen. A curved spline is then drawn from the source to the destination, and an arrowhead is drawn at the receiving end. The size of the arrowhead varies so that it is never bigger than the destination. The splines are defined by seven points (called *knots*) placed along the intended path. Three of these define the exit point and direction from the pointer and three define the entrance point and direction to the referent (see Figure 5.9). A final knot is added in between to make the arrow path smoother. Three points are used at each end to allow the arrow to leave the pointer from any side and intersect the referent at any point.

Splines are used rather than a straight line since it is more attractive and does not cause confusion with other lines in the picture (no "collision avoidance" is done). In addition, it is simpler to draw a curved line since the arrows can always be drawn leaving the pointer from the right and intersecting the referent on the left (Figure 5.9). The DrawArrowFrom procedure returns information needed to erase the arrow. This is then stored as part of the display data in the pointer document.

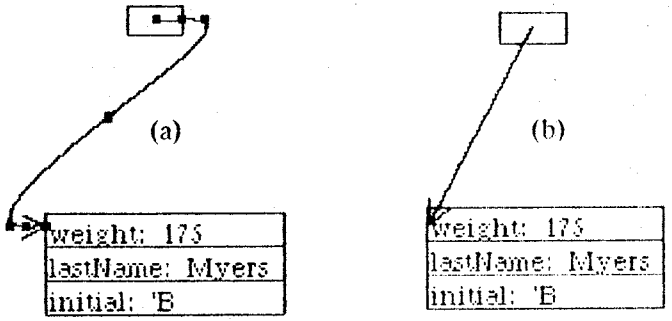


Figure 5.9. Demonstration of the advantage of curved lines used in Incense (a) over straight lines (b). The knots used in drawing the spline are shown as black squares in (a).

5.3.3 Erasing a document.

After a document is displayed, it useful to be able to make it disappear. The process of removing the picture from the screen is called *erasure* (as opposed to *de-allocation* or *destruction* of the actual document itself which is a separate operation). An erased document can be re-displayed with a new size, location and format. The erase procedure of a document, as well as all the other

non-display procedures, operate similarly to the formatProc, in that its main purpose is to pass control to another erase procedure at the subformat level. This is necessary since the different subformats may have caused different displays to be used. For example, to erase a record, each field document must be erased, but one subformat may have omitted a field due to the lack of sufficient room, whereas another subformat would have included it. The top level erase procedure simply calls the low level (or *internal*) erase procedure corresponding to the subformat that was used for the current display.

5.3.4 Selecting a document.

Once documents have been displayed, some way is needed to refer to them. The user may want to modify, re-display, or change the format used for a document. In Incense, the user refers to documents simply by pointing at them using the mouse. The document referred to is said to be *selected.* There can be only one document selected at a time and its picture is video reversed on the screen (in a similar manner to DLISP and Smalltalk). This is not sufficient, however, since documents can contain other documents. For example, a record document contains documents for the various fields. In order to allow any of these to be selected, Incense always selects the smallest (in screen area) object under the mouse. If this document is already selected, however, its parent is selected instead. All documents fit into a hierarchy where the top document was displayed by the user and lower level documents are displayed as subordinates of aggregates or pointers.

Incense has a special procedure that returns the selected document so that it can be manipulated. All documents accept the message *FindSelection,* which has the coordinates of a point as a parameter. It is intended that these coordinates come from the position of the mouse, but other options are allowed. When FindSelection is called, the document determines whether the point is inside its current screen picture (it is an error to call this procedure for a document which is not displayed). If not, the procedure returns *missed.* Otherwise, if the document is already selected, it is de-selected and then the procedure returns *next.* This indicates to the parent that it should be the selected document. If neither of these conditions is true, the procedure will test each of the document's subordinates, if any, to see if they want the selection. If any of them returns *next* or if none want the selection, then *hit* is returned and the current document is selected. See Figure 5.10 for an example of selections moving up the document hierarchy.

There is a special JAM function that cycles through all of the documents on the display testing calling FindSelection on each. The user can click the red button to select documents until the correct one is found. A click on the yellow-button will then cause that document to be erased. Then the mouse is used to specify a rectangle in which the erased document will be redisplayed. This is very useful for expanding documents displayed too small to see (Figure 5.11). The documents explicitly displayed by the programmer (called *top-level*) are *tagged* (in the AMBIT/G sense) in that they are the starting points for all searches. Thus, the selections to move up the

hierarchy in the correct manner. Another feature is that the more recently displayed top-level documents (which tend to be smaller in practice) are searched first. This allows a record placed on top of a layout, for example, to be selected before the layout underneath. This user interface is very natural, making exploration of multi-level structures fast and easy.

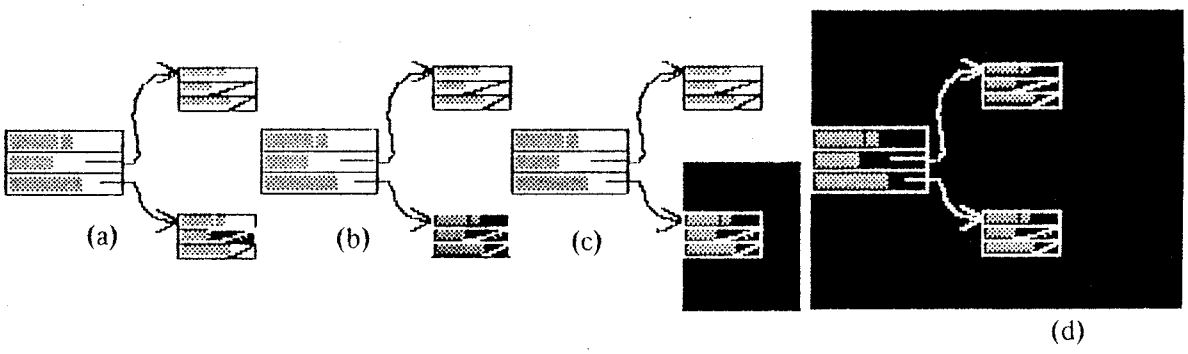


Figure 5.10. Selections moving up the document hierarchy: from record field (a) to record (b) to layout for record (c) to layout for everything (d).

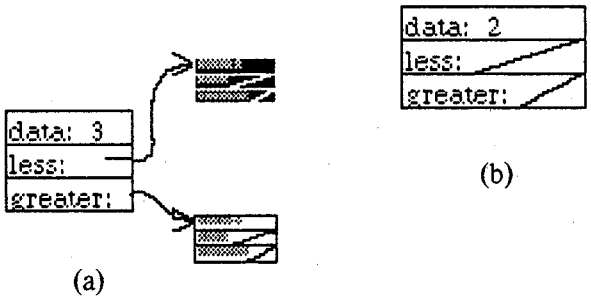


Figure 5.11. The display for the selected record in (a) expanded in (b).

5.3.5 Editing a document.

In addition to erasing and redisplaying a document, modifying the actual value of the associated data structure is useful. Incense currently only provides the structure for this operation. In future versions, to modify variables of the basic types such as INTEGERS, REALS, and BOOLEANS, the user will type a new value. For pointers, however, the user will be allowed to specify the new value simply by pointing at the new referent on the display. The pointer document will then do all the calculations necessary to deduce the correct value to be stored as the pointer's value. Type-checking of the pointer and the referent will be done to assure legality of the modification. For clocks, as another example, the user will be able to select a hand and rotate it to the correct value. Thus, the result of the editing operation will be vastly different depending on the particular format used to display the data. The edit command in Smalltalk handles this problem by taking no

arguments. The system is put into a state where all user actions are handled directly by the edit procedure, but Incense will probably use a different approach (see section 8.1.3).

5.3.6 Deallocating a document.

Current Mesa does not have any built-in storage management. It is therefore necessary for programs to handle deallocation of storage by themselves. Incense, as it may be clear, requires a large number of variable-length structures and consequently allocates storage for them. To allow this storage to be re-used, a *destroy* procedure has been added to documents. A great deal of sophistication would be needed to find all of the storage actually used by a document since there are many places where arrays of pointers are used, some of which may point to the same place. Deallocating something more than once in Mesa causes the entire system to crash. This problem will disappear in Cedar, however, with the advent of an automatic garbage collector. Therefore, little effort was expended to perfect the destroy procedures and they occasionally fail.

VI. CedarSymbols: The Type System for Incense

To automatically generate documents and forms for Mesa data structures, Incense requires the ability to ascertain the types, memory addresses, and values the data to be displayed. As discussed in section 4.2.1.1, the compiler produces detailed symbol tables that contain sufficient data to get this information for any variable in a program. The existing facilities for accessing this information, however, were not sufficiently modular, efficient and extensible to be used for this application. The current Mesa debugger, for example, has the symbol table manipulations tightly coupled with storage management, interpreter, and user interface mechanisms. It was therefore necessary to design and implement a system for accessing the symbol tables as part of the work of Incense. This chapter describes the design for this system, which is called *CedarSymbols*.

6.1 Goals of CedarSymbols.

It was clear that a system such as CedarSymbols would be needed in Cedar, and some attempt was made to include the requirements of Cedar in the design. The current implementation of CedarSymbols, however, will need to be revised before being used by any production system. This section discusses some of the design considerations that affected CedarSymbols.

6.1.1 Opaque types.

The amount of information needed to define many data types in Mesa is quite large. For a record, the programmer needs to know the number of fields, the field names, and the types of each field, for example. The symbol tables contain all this information in a highly encoded form which is inconvenient to access. While a copy could be made that was simpler to use, large amounts of memory and time would be wasted since the information existed in memory already and much that was translated might not be needed. It was therefore decided to hide the details of the implementation by using *opaque* types. The internal data and structure of the types is not available to the client since all information about the types is obtained through standard procedures.

It was decided early that some of the principles of data abstractions would be important for CedarSymbols. The design is not completely object oriented like the design for documents, however, mostly due to the difficulty in current Mesa of having different objects accept different messages. The data structures used to represent a type are made completely opaque, however, through the use of a POINTER TO UNSPECIFIED, called a *TypeID*. A procedure called *GetType* can be

used to discover the basic type of a TypeID. GetType returns an element of an enumerated type (called *Types*) containing all the legal types:

```
Types: TYPE = {noType, unspecified, integer, real, cardinal, boolean, character, word, string,
    enumerated, subrange, pointer, union, record, array, descriptor, signal, error, condition,
    process, procedure, userDefined, typeType};
```

6.1.2 Opaque memory addresses.

A large amount of information is required to describe the memory location of data. A simple POINTER is not sufficient because Mesa has types which require different amounts of storage. INTEGERS, BOOLEANS, and CARDINALS normally are stored in one word each, but if they are embedded in a PACKED ARRAY they use 1, ½, and 1 word respectively (and if in a PACKED RECORD, 1 word, 1 bit, and 1 word). In addition, there are LONG types which are stored using 2 words, as are REALS. PACKED RECORDs add a further problem in that the various fields can start and any arbitrary bit position. For example,

```
messyRec: PACKED RECORD
                [
                a: [1..7],              --(0, 3)
                b: CHARACTER,           --(3, 8)
                c: [-3..0],             --(11, 2)
                d: BOOLEAN,             --(13, 1)
                e: [20..22],            --(14, 2)
                ];
```

fits exactly into one Alto word (16 bits). Each field above starts at the bit position x, taking y bits, in the (x, y) comment following the field.

The opaque type *MemoryAddress* in CedarSymbols abstracts out much of this complexity. As with TypeIDs, the client is given a POINTER TO UNSPECIFIED from which he can only get information by calling CedarSymbols routines. At some point the client will need to get at the actual data. There could have been a different procedure for each type, but instead two generic functions were provided. *GetOneWordValue* has as a parameter a MemoryAddress. It extracts the value from MemoryAddresses whose lengths are 16 bits or less. The function copies the data into the appropriate part of the resultant word while masking out all other bits. Thus, while the memoryAddress for messyRec.d will specify the actual location in memory, GetOneWordValue called with that memoryAddress as an argument will return a word that can be used as a normal BOOLEAN. There is also a PutOneWordValue routine that stores values in the correct format. Another pair of routines is provided for two-word values. No non-aggregate type in current Mesa is larger than two words. Decomposing aggregate structure is done through the use of the *AddrOfSub* routine described in section 6.3.

Another advantage of opaque memoryAddresses is that they can be constructed for constants simply by allocating some storage for the value. The symbol tables contain sufficient information for reconstructing many constants, so Inconse allows the user to request a display for a value such as maxNumEls: CARDINAL = 15; whereas the current debugger does not. The memoryAddresses for constants are flagged *readOnly* so any attempt to store into one causes a signal. The memory for the value is deallocated automatically when the memoryAddress is.

6.2 TypeOfSub.

An early design for CedarSymbols provided a separate procedure to handle every type. This required a large number of procedures, so one generic procedure is used instead. *TypeOfSub* takes a typeID and an index and returns a typeID representing the type of the subcomponent. Since the generic procedure is implemented using a case statement branching to a different procedure for each type, it would be easy to provide both interfaces, but there seems to be no motivation for this. The interpretation of the index argument to TypeOfSub depends on the specific type of the typeID. If no action is specified for a particular index, or if it is out of bounds, a signal is raised. In particular: (in the following, *<Name>* will mean a typeID for *Name*.)

Subrange: For subrange types, if index = 0 then the type over which the subrange occurs is returned. Example, for `Sub1: TYPE = [1..10];` `TypeOfSub[<Sub1>, 0]` returns a CARDINAL typeID.

Pointer: For pointer types, if index = 0 then the type of the referent is returned. Example, for `p1: POINTER TO BOOLEAN;`

`TypeOfSub[<p1>, 0]` returns a BOOLEAN typeID.

Array: For array types, if index = 0 then the type of the array index (which will be a subrange or enumerated type) is returned. For index = 1, the type of the elements is returned.

Array Descriptor: For an array descriptor types, if index = 0, the type of the array described is returned.

Record: For a record type, the type of the indexth field is returned. Example, for `r1: RECORD [f0: BOOLEAN, f1: CARDINAL];` `TypeOfSub[<r1>, 1]` returns a CARDINAL typeID (counting is from zero).

Union: Unions types are used for the variant parts of records. Thus for a variant record, the type of the entire variant part is *union*. TypeOfSub applied to a union typeID returns the type of the tag for index = 0. Since the actual value of the tag is needed to find which variant is current and thus the types of the fields of the variant part, a special routine is needed (see section 6.4.5).

Transfer: Transfer types include PROCEDURES, SIGNALS, ERRORS, PROCESSes, AND PORTS. These all have arguments and return values that are represented as records. Therefore, TypeOfSub with index = 0 returns a record type describing the arguments. With index = 1, it returns a record describing the return values.

TypeType: Mesa allows the programmer to define new types which can then be used in the definition of variables. CedarSymbols allows the client to discover this information. TypeOfSub on a typeType typeID returns the base type if index = 0. Example, for

```
Age: TYPE = POINTER TO CARDINAL;
```

`TypeOfSub[<Age>, 0]` returns a POINTER typeID. Note that typeIDs of type typeType never have associated memoryAddresses.

UserDefined: Once a type has been defined, any variables defined using that type have type *userDefined* in CedarSymbols. The Mesa compiler does not use this distinction, but it is important for Incense to be able to assign a particular display, for example, for Age, and not have it apply to all POINTER TO CARDINALS. TypeOfSub returns the base type for index = 0. Variables of type userDefined do have memoryAddresses. For example, for

```
myAge: Age ← 22;
```

myAge has type userDefined and TypeOfSub of the associated typeID would return a POINTER typeID.

6.3 AddrOfSub.

As described in section 6.1.2, Get*X*WordValue (where *X* = *One* or *Two*) is used to get the value of abstract memoryAddress for variables of the basic types. For aggregate types, however, decomposition must be done first. *AddrOfSub* provides the required capability in the same manner as TypeOfSub. In particular:

Pointer: The MemoryAddress of the referent is returned for index = 0. This allows the client to avoid having to know the details of how the pointer is stored.

Array: The address of the indexth element of the array is returned.

Array Descriptors: An array descriptor is actually a record containing a POINTER TO ARRAY and a CARDINAL specifying the length of the array. AddrOfSub with index = 0 returns the address of the POINTER part of the descriptor, with index = 1, returns the address of the length (CARDINAL) part, and for index = 2, returns the address of the array itself.

Record: The address of the indexth field of the record is returned.

Union: The address of the tag is returned for index = 0. For index = 1, the address of the entire variant part is returned. Note that the length in the memoryAddress for the variant part will be determined by the current value of the tag if the different variants have different lengths.

6.4 Other Routines Needed by Certain Types.

Mesa is a very complex language and the generic procedures described above do not provide all of the information required for all types. Therefore, some additional procedures are provided to handle specific problems. These also were made as generic as possible.

6.4.1 Index <--> Name.

For records and enumerated types, it is necessary to translate between the actual string names used in the program and the indices required in the above routines. *GetIndexFromName*, which

accepts a typeID and a string, searches for a field for records or a name for enumerated types which uses that string as the name. It then returns the corresponding index or raises a signal if it is not there. *GetNameFromIndex* translates in the opposite direction.

6.4.2 Maximum Index.

It is useful to be able to get the maximum legal index that can be used in the CedarSymbols procedures for array, record, enumerated and subrange types. *GetMaxIndex* takes a typeID and returns the maximum index (which is one less than the number of legal values since counting is from zero). For arrays, this operation is simply: `GetMaxIndex[TypeOfSub[arrayTypeID, 0]]`.

6.4.3 Subrange types.

Subrange types have three special procedures associated with them. *GetOrgRangeSubrange*, which takes a subrange typeID, returns the lower bound (*origin*) and the maximum legal index (*range*). The origin is an INTEGER to allow subranges of INTEGERS, but will be non-negative for subranges of CARDINALS and enumerated types. The range is the number of elements in the subrange *minus one* (since the indices start at zero). The other two procedures provided are similar to the index and name translation procedures, since they allow the conversion of an index to a value used in the program. In this case, the value is the actual number stored in memory. With subrange types, the lowest legal value is always represented in memory by zero, so the client would have to add the origin to find the number used in the program text. *GetIndexFromValue* takes a typeID and the value stored and returns an index suitable for displaying or using to get the name for enumerated types. *GetValueFromIndex* translates in the opposite direction. For example, for:

```
Days: TYPE = {Sun, Mon, Tues, Wed, Thurs, Fri, Sat};
WeekDays: TYPE = Days[Mon..Fri];
today: WeekDays ← Tues;
```

the value stored in `today` would be `1`, and GetIndexFromValue can be used to translate that to the `2` that GetNameFromIndex would take to return `"Tues"`. Note that these two procedures could be computed by the client using GetOrgRangeSubrange.

6.4.4 Procedure types.

The value stored in procedure variables is a tightly coded representation which includes a pointer to the code to be executed. One useful piece of information about procedure variables is the name of the constant procedure that was assigned to this variable. For example, for:

```
MyProcedureType: TYPE = PROCEDURE [arg1: CARDINAL] RETURNS [ret1: BOOLEAN];
Proc1: MyProcedureType = BEGIN . . . END;
ProcVar: MyProcedureType ← Proc1;
```

where MyProcedureType is a type, Proc1 is a constant procedure of that type, and ProcVar is a procedure variable of that type. *GetProcedureName*[<ProcVar>] will return the string "Proc1" which is the name of the procedure that would actually be executed if ProcVar was called. If Proc1 had been a non-constant procedure, initialized with "←" instead of "=":

```
Proc1: MyProcedureType ← BEGIN . . . END;
```

then there is no constant procedure that can be found that names the code between the BEGIN and END (since Proc1 is now a variable and could be assigned another value). In this case, the string "<ANONYMOUS>" would be returned.

6.4.5 Union types.

For *union* types, unlike with records, the actual value of the tag is required to discover the subtype . This is the only situation in Mesa where the type of something depends on its value. *GetUnionVariantType* takes a union typeID and a memoryAddress and returns the type of the variant part. This is a record TypeID, and the standard routines can be used on it.

6.4.6 UserDefined and TypeType types.

While it is important to distinguish userDefined types from the types they are based on, the client frequently is not interested. The *ToBase* procedure is therefore provided. It converts a typeID into a base type. If the argument typeID is not userDefined or typeType, then it is simply returned unmodified. ToBase uses no internal information and could have been written by the client.

6.5 Contexts.

The reader may have noticed that no way has been described for getting the *first* typeID and memoryAddress from which to find the subcomponents. This is handled through the use of *Contexts*. A context describes a module, procedure, or block that contains definitions of variables or types. One procedure returns the context for a module of a certain name. Another returns the context for the procedure on the top of the execution stack for a particular process. Other procedures allow iteration through contexts in execution call order or lexical ordef. This allows traversal to be either top-down (*e.g.*, from the module to all of its components) or bottom-up (*e.g.*, from a procedure to its callers). Finally, there are procedures for getting the typeID and memoryAddress for a context. The type of the typeID will be a record, and the memoryAddress only exist for contexts describing modules, procedures or blocks that currently are active (since only then do they have memory assigned to them).

VII. Incense – Details of the Implementation

This chapter describes some of the details of the implementation of Incense. Section 7.1 contains a discussion of exactly what documents are and how they work. Following that, some details are given about how documents are displayed: section 7.2, erased: section 7.3, and created: section 7.4. Section 7.2 discusses the basic documents, records and layouts in detail. Arrays are so similar to records that they do not require any further elaboration. Section 7.2 also describes the built-in facilities for allowing the client to specify the rectangles for field placement. Finally, section 7.5 discusses how a client could define his own documents.

7.1 Implementation of Documents.

Mesa currently does not have convenient methods for supporting object-oriented programming. The motivation for this style out-weighed the problems, however, so a somewhat clumsy scheme was used. In order for this to work, LOOPHOLEs had to be used to breach the type system. A document creator stores into UNSPECIFIED pointers in the document whatever data is necessary. Since the procedures that manipulate this data are attached to the document, they can always know the type of the data. A well-formed document therefore has information about the type of the data structure to be displayed embedded in the **typeID** field, in the type of the various data fields, and in the specific procedures themselves. These must be all consistent for Incense to operate correctly.

The definition of a document and its constituents is given below and Incense's picture of a typical document for an INTEGER is given in Figures 7.1 and 7.2. Note that DESCRIPTOR FOR ARRAY is used to implement an array of a variable size.

```
Document: TYPE = POINTER TO DocumentRecord;

DocumentRecord: TYPE = RECORD
    [
    formatSet: FormatSet,
    procs: pProcedures,
    typeID: TypeID, --of associated data to be displayed
    addr: MemoryAddress,    --of associated data to be displayed
    parent: Document,
    displayed: BOOLEAN,     --The following are only valid if doc is displayed
    displayUsed: SubformatIndex, --is index used to display doc;
    selected: BOOLEAN,
    myAbsPos: Rectangle     --position on screen
    ];

--A formatSet is an array of formats, each of which in turn is an array of subformats.  If f is a FormatSet, then
    f.formats[x].formatProc[args] is used to display the datum in format x.  The subformats should not be
    accessed directly.
FormatSet: TYPE = RECORD
    [
    formats: DESCRIPTOR FOR ARRAY OF Format,
    data: POINTER  --global data useful for displaying document in all formats.  This is one place the type
                       system is breached.
    ];
```

```
Format: TYPE = RECORD
    [
    formatProc: FormatProc,
    subformats: DESCRIPTOR FOR ARRAY OF SubformatInfo --internal
    ];

SubformatInfo: TYPE = RECORD
    [
    subformatProc: SubformatProc,
    iprocs: pInternalProcs, --procedures for this subformat
    subData: POINTER --other useful data for display specific to this subformat. This is one place the type
                     system is breached.
    ];

FormatProc: TYPE = PROCEDURE [me: Document, formatIndex: FormatIndex, maxArea: Rectangle]
    RETURNS [areaUsed: Rectangle];

SubformatIndex: TYPE = RECORD
    [
    format: FormatIndex,
    subformat: FormatIndex
    ];

FormatIndex: TYPE = CARDINAL;

SubformatProc: TYPE = PROCEDURE [me: Document, myIndex: SubformatIndex, maxArea: Rectangle]
    RETURNS [used: BOOLEAN, areaUsed: Rectangle];
      --if applicable then used = TRUE and executes;
      --otherwise used = FALSE

pProcedures: TYPE = POINTER TO Procedures;

pInternalProcs: TYPE = POINTER TO InternalProcs;

Procedures: TYPE = RECORD
    [
    destroy: Destroy,     --erase if displayed and de-allocate storage
    erase: Erase,     --to un-display doc; doc not destroyed
    findSelection: FindSelection,
    deSelect: DeSelect,
    findField: FindField, --for layouts
    findDocUnder: FindDocUnder,     --for layouts
    drawArrowFrom: DrawArrowFrom, --for layouts
    edit: Edit
    ];

InternalProcs: TYPE = RECORD
    [
    destroy: IntDestroy, --erase if displayed and de-allocate storage
    erase: IntErase, --to un-display doc; doc not destroyed
    findSelection: IntFindSelection,
    deSelect: IntDeSelect,
    findDocUnder: IntFindDocUnder,    --for layouts
    drawArrowFrom: IntDrawArrowFrom,     --for layouts
    edit: IntEdit
    ];

Select: TYPE = {hit, next, missed};

Destroy: TYPE = PROCEDURE [me: Document]; --erases and de-allocates document and any subdocuments

Erase: TYPE = PROCEDURE [me: Document, eraseScreen: BOOLEAN];

FindField: TYPE = PROCEDURE [me: Document, ptrID: PtrID, dataPointedTo: MemoryAddress, dataType: TypeID]
    RETURNS [fieldDoc: Document];

DrawArrowFrom: TYPE = PROCEDURE [from: ThreePoints, me: Document] RETURNS [pts: SevenPoints, destHeight:
    Coordinate];
        --returns all points of spline to allow erasure

--The following finds a doc under or equal to me that has data as its data of type dataType.
FindDocUnder: TYPE = PROCEDURE [me: Document, data: MemoryAddress, dataType: TypeID] RETURNS [dataDoc:
    Document];
```

```
FindSelection: TYPE = PROCEDURE [me: Document, mouse: Point] RETURNS [sel: Select, selD: Document];
    --if mouse point is for this object, selects self and returns hit unless
    --already selected, in which case, returns next and de-selects self,
    --otherwise, returns missed.

DeSelect: TYPE = PROCEDURE [me: Document];

Edit: TYPE = PROCEDURE [me: Document, newValue: MemoryAddress];

IntDestroy: TYPE = PROCEDURE [me: Document, myIndex: SubformatIndex];

IntErase: TYPE = PROCEDURE [me: Document, eraseScreen: BOOLEAN, myIndex: SubformatIndex];

IntDrawArrowFrom: TYPE = PROCEDURE [from: ThreePoints, me: Document, myIndex: SubformatIndex] RETURNS
    [pts: SevenPoints, destHeight: Coordinate];

IntFindDocUnder: TYPE = PROCEDURE [me: Document, data: MemoryAddress, dataType: TypeID, myIndex:
    SubformatIndex] RETURNS [dataDoc: Document];

IntFindSelection: TYPE = PROCEDURE [me: Document, mouse: Point, myIndex: SubformatIndex] RETURNS [sel:
    Select, selD: Document];

IntDeSelect: TYPE = PROCEDURE [me: Document, myIndex: SubformatIndex];

IntEdit: TYPE = PROCEDURE [me: Document, newValue: MemoryAddress, myIndex: SubformatIndex];
```

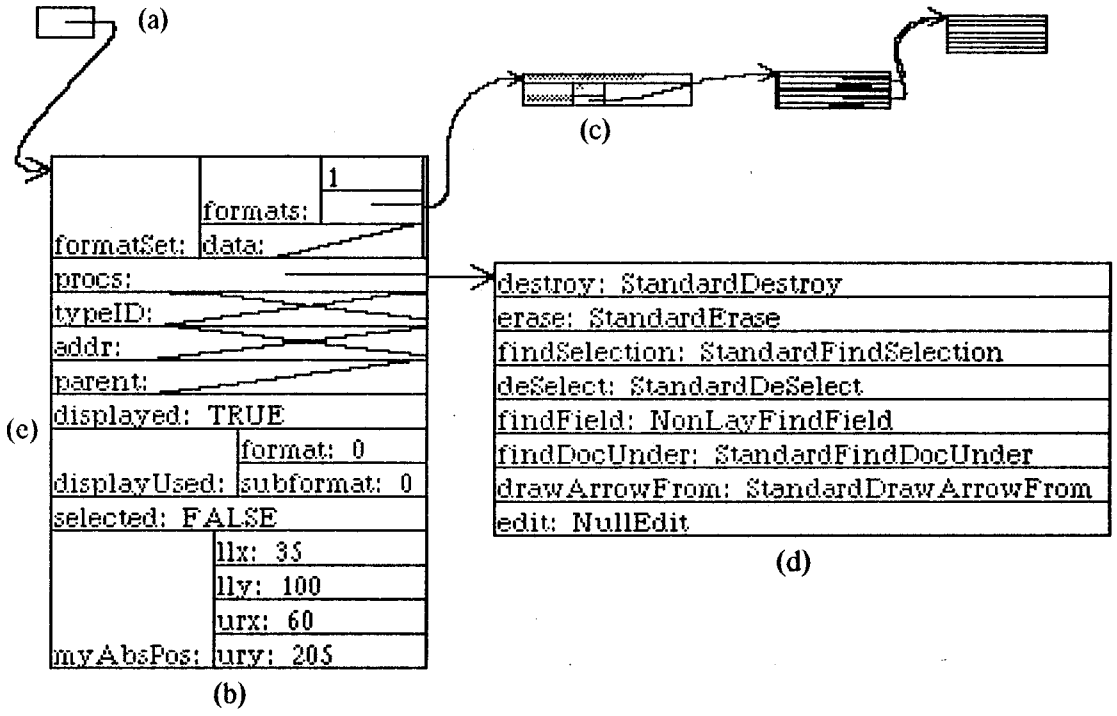


Figure 7.1. Incense display of a document for a displayed INTEGER. The document (a) is a pointer to a documentRecord (b). That record contains 1 format (c) expanded in Figure 7.2, a set of procedures (d), and other data (e).

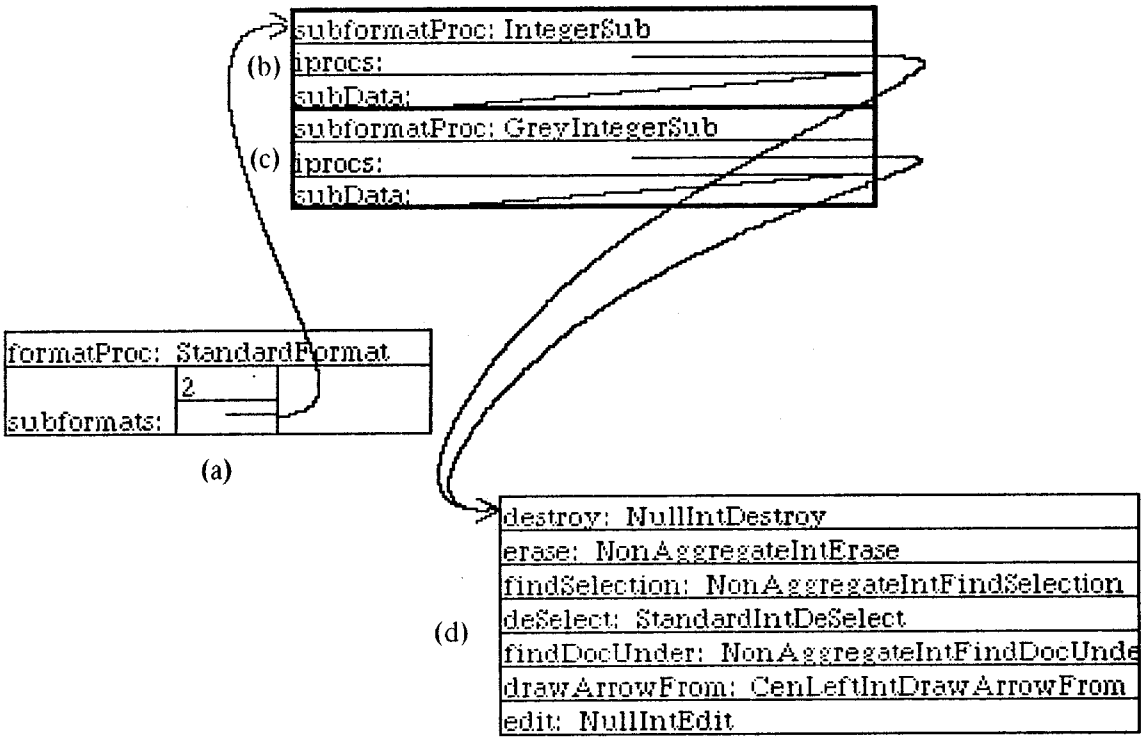


Figure 7.2. Expansion of integer document format. There is one format with its formatProc: (a) and two subformats: (b) and (c). The subformats have different subformatProcs and no subData, but they share the same internal procedures: (d).

The fields of a **documentRecord** are:

formatSet which contains the procedures and data required for displaying;

procs which are the top level procedures for doing all of the other operations on documents such as erasing, selecting, etc. These typically call on the corresponding **internalProc** in the **subformatInfo** for the current subformat;

typeID and **addr** for the data structure this document is meant to display. For some documents, such as those for layouts, these fields will be NIL since there is no associated data;

parent which allows tracing up the document hierarchy. The parent field is used by procedures such as FindField which must search up the tree. Every document has at most one parent. Some documents that have subordinates, such as for the fields of records. The information describing the subordinate documents is stored in the **subData** field of **subformatInfo**;

displayed which tells if this document is displayed. The rest of the fields in a document are only valid if the document is displayed;

displayUsed contains the format and subformat indices used to display the document. This is required by the procedures that manipulate the displayed documents since they need to know which **internalProc** to use;

selected tells whether this document is the one selected or not;

myAbsPos gives the rectangle that the document fit into when displayed. It is guaranteed to be less than or equal to the rectangle specified as **maxArea** in the **FormatProc** used to display the document.

7.2 Displaying Documents.

As described in section 5.3.1.1, a client causes a document to be displayed by choosing a format and calling the associated **formatProc**. This formatProc iterates through the subformatProcs to find one that will display the document. The operation of the subformats for various types is described below. In addition, however, the formatProc takes care of other bookkeeping tasks. First, if the document was selected, it is deselected. Next, the **displayed** flag is set to true and **displayUsed.format** is set. After the subformats are checked, the document is *registered.* A list is kept of all the documents that are currently displayed, and a *RegisterDoc* procedure is provided for adding documents to the list. This list allows the erasure of all documents on the screen (and a JAM function provides this operation to the user). In addition, the list is necessary for the selection process and for allowing multiple pointers to the same data structure to point to the same place on the screen. A generic procedure *EnumerateRegDocs* is provided:

```
EnumerateRegDocs: PUBLIC PROCEDURE [proc: PROCEDURE [enumD: Document, enumTopLevel: BOOLEAN]
    RETURNS [BOOLEAN]] RETURNS [docChosen: Document];
```

where the proc argument is a procedure to be called on each registered document. EnumerateRegDocs terminates and returns the current enumD argument when proc returns TRUE. The enumTopLevel flag is used to distinguish documents that were explicitly displayed by the client from those displayed by other documents. The formatProc procedure sets the topLevel flag FALSE, and the utility routine *DisplayDoc* called by the client changes it to TRUE for the appropriate documents.

7.2.1 Displaying the basic types.

Documents for data of types STRING, INTEGER, CARDINAL, WORD, UNSPECIFIED, BOOLEAN, CHARACTER, REAL, PROCEDURE, and ENUMERATED do not need any extra information for display. Subranges of these do store the type of the base and the offset and range of the subrange for efficiency, however. There are four subformats defined for each of these types. Two of these display the data using text and two display the data using grey blobs. One of each of these draws a box around the value and the other does not. The choice between the "boxed" or "non-boxed" subformats is usually made by the creator of the document based on whether it will be displayed in an aggregate structure or not. For example, the record document will draw a box around the entire field and not just the value portion so the unboxed option is chosen for the values enclosed. In

Figure 7.3, the value in a field has been selected showing the extent of its rectangle. A box is drawn around documents of the basic types that are intended to be displayed by themselves (see Figure 5.4). This is especially useful if there is a pointer to the value (Figure 7.4).

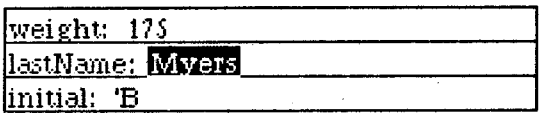


Figure 7.3. Selection of a field value in a record showing the extent of its rectangle.

Figure 7.4. Pointer to CARDINAL showing utility of the boxes.

The documents for the basic types contain subformats for normal and grey display. One of these styles is chosen at run time by the subformats based on the amount of room available (**maxArea**). The normal subformat will not be used if the area specified is smaller in the vertical ("Y") direction than the height of the font used to display the value, or if less than half of the value will fit in the horizontal ("X") direction. The latter restriction is used since half of the string displayed is frequently enough to allow recognition of the value (see Figure 7.5a and 7.5b). The grey blobs used if there is not enough room are supposed to represent very tiny text. Thus, the size of the blob is adjusted to correspond to the length of the string that would have been printed for the value. This may allow the programmer to distinguish between a number of values if they have different lengths (see Figure 7.5c and 7.5d).

Figure 7.5. Demonstration that clipped strings do supply information: (a) and (b) are values of BOOLEANS. Grey areas are different sizes depending on string: (c) is for FALSE and (d) is for TRUE.

Two utility procedures are used by these subformats. One displays a string in a given rectangle, clipping it if necessary. The other displays a grey blob of a given height and width. This makes it easy to add subformats for displaying new types whose values can be represented as simple strings. For example, PROCEDURE types recently were added to Inconse in just a few minutes after GetProcedureName was added to CedarSymbols.

7.2.2 Displaying records.

Records are more complex than INTEGERs and require extra information to be displayed. No global data (**formatSet.data**) is needed, but a great deal of information is stored in the **subData** slot for each subformat. This data includes two documents for each record field. This is an example of recursive nesting of documents since any field may be a record. For the automatically created record documents, the **subData** is the same for both subformats, so the **subData** pointers refer to the same data.

7.2.2.1 Mesa definition of record document.

The definition for the data structure used in the **subData** field of records is given below:

```
pRecSubformatData: TYPE = POINTER TO RecSubformatData;
RecSubformatData: TYPE = RECORD
    [
    needed: Rectangle,  --area needed for this subformat to work;
            --must be bigger that sum of field rects (in base)
    maxNameWidth: Coordinate,  --needed to see if will fit (in base)
    arrowEnd: ThreePoints,  --dest point for arrow (in local coord sys)
    curArrowEnd: ThreePoints,  --set when displayed (in Base coord sys)
    fields: DESCRIPTOR FOR ARRAY OF RecFieldData
    ];
RecFieldData: TYPE = RECORD
    [
    dataDoc: Document,      --for actual data contained in field
    dataForInd: FormatIndex,
    dataRect: Rectangle,  --relative to needed rect
    nameDoc: Document,      --for field name
    nameForInd: FormatIndex,
    nameRect: Rectangle  --relative to needed rect
    ];
```

The fields of **RecSubformatData** are:

needed, a rectangle describing how big the record display was when it was designed. For the basic types, such as BOOLEAN the size is easily calculated from the current value. This is not true for records and other aggregate structures that can have fields in arbitrary locations (see Figure 5.8). Therefore, the size is stored as data. The rectangle is calculated in a base coordinate system so that changes of scaling will not affect its size;

maxNameWidth, the width of the longest field name. This is used to decide if the rectangle supplied will be large enough to display a reasonable amount of the record (see below);

arrowEnd, the place on the record where an arrow drawn to the record should end. The basic types simply use the center of the left side, but for aggregate structures, it is useful to be able to specify the destination points. Three points are used to allow specification of the direction as well as the position of the arrow;

curArrowEnd, the **arrowEnd** points converted so they correspond to the current position of the record; and

fields, an array of data needed for each field.

The extra data needed for each record field is:

dataDoc, the document which will display the value of this field;

dataForInd, the formatIndex to be used when displaying that field's value document;

dataRect, the rectangle to be used to hold the value;

nameDoc, a document used for displaying the name of the field. The name was made a document to allow more consistent handling of the name and the value portions of the field display. The **nameDoc** usually is a simple string-handling document; and

nameForInd and **nameRect**, the formatIndex and rectangle for the name part of the field display. Note that this allows the field name and field value to be displayed in any position, not just the *name: value* as used in the default. Thus, for example, a special record field name document might be created that centered the name in the field or put it flush with the top.

7.2.2.2 Operation of the subformats

The default record documents have one format and two subformats. As with basic types, a subformat is chosen based on the amount of screen area available. For records, the **needed** and **maxNameWidth** fields of the **RecSubformatData** are used in this decision. The standard subformat is used if the **maxArea** is larger in the vertical direction than the height of the **needed** rectangle, and if it is wider in the horizontal direction than **maxNameWidth**. This allows part of the value to be clipped (Figure 7.6). If there is not enough room, the other subformatProc is used. This latter procedure introduces a scaling factor that allows the entire **needed** rectangle to fit into the area specified. The proportion of the height to width of the resulting rectangle on the screen is kept the same as the proportion in the **needed** rectangle irrespective of any uneven scaling that might have been imposed from above. This scaling must be taking into account, however, since it affects the size of the **maxArea** rectangle. The formula used to determine the scaling amount is:

```
scale ← MIN[maxAreaWidth/neededWidth, maxAreaHeight/neededHeight];
```

where both maxArea and needed have been converted into the same (*base*) coordinate system.

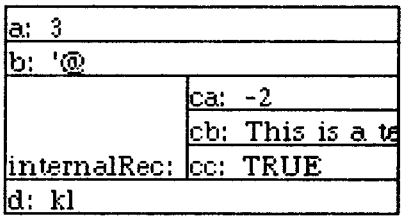


Figure 7.6. Record with a value clipped. Most of the information is still available.

A further complication arises since the record display should be centered vertically inside the **maxArea** rectangle. This is straightforward in the non-scaling case, but here it requires the scale factor to be taken into account. The formula for the starting Y position is:

```
yStart ← MAX[maxAreaLowerY, (maxAreaHeight-neededHeight*scale)/2];
```

Figure 7.7 shows a scaled record centered inside of the bounding box and taking the same proportions as the full-size display.

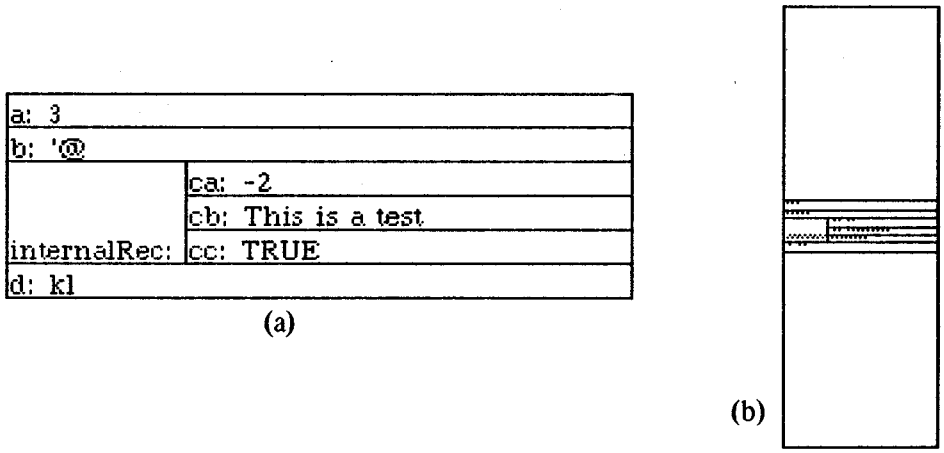


Figure 7.7. Record shown full size (a), and scaled proportionally and centered inside a bounding rectangle (b).

After all these manipulations have taken place, both the record subformats first calculate the destination points for arrows in the current context. If the record contains a pointer back to itself, it will therefore be drawn to the correct place. Afterwards, the subformats simply iterate through the fields causing the name and field documents to be displayed in their respective rectangles.

7.2.2.3 Display of clocks.

Clocks are record documents with two special subformat procedures. Clock documents have two formats, one of which displays as a normal record and the other as a clock with hands (Figure 5.1). The clock format differs from the others discussed in that it does not decide which subformat

to use based on **maxArea**. The first subformat is used if the document is being displayed for the first time. If the document is being re-displayed with the same rectangle as used previously, however, the second subformat is chosen which knows how to move the hands on the existing display. Thus, the clock document can be associated with a process that will cause it to be re-displayed when the time has changed, and a simple JAM procedure to do this has been written.

The angle of the hands of the clock are easily calculated as follows:

```
minAngle ← -(min*6 - 90 + (sec/10));
hourAngle ← -(hour*30 - 90 + (min/2));
```

Thus the hands can move continuously rather than in unit steps.

7.2.3 Displaying layouts and pointers.

As mentioned in section 5.3.1.3, the display of layouts and pointers is very complex. Since no space allocation is done at run-time, the layout documents, like records, contain rectangles specifying where each field should be placed. These rectangles are allocated in a very simple manner described in section 7.4.7. This section discusses some of the data structures and procedures that are used by the pointer and layout documents to decide what fields to use and to draw the arrows.

7.2.3.1 Mesa definitions for pointers and layouts.

In section 5.3.1.3, it was mentioned that layout documents contain a document for each field. The following are the definitions for the data structures needed for the **data** and **subData** fields for pointer, layout and layout field documents:

```
--Pointers:
--PtData fits in the data slot of FormatSets since is global; pPtSubformatData goes in the subformat subData slot
pPtData: TYPE = POINTER TO PtData;
PtData: TYPE = RECORD
    [
    subDataType: TypeID,          --need type to create dataDoc if not available
    subDataAddr: MemoryAddress,
    ptrID: PtrID --used to associate pointer with a layout field
    ];
pPtSubformatData: TYPE = POINTER TO PtSubformatData;
DisplayRec: TYPE = RECORD
    [
    splinePointsUsed: BOOLEAN, --false if not (e.g. due to NIL)
    splinePoints: SevenPoints, --kept to allow erase
    destHeight: Coordinate, --kept to allow erase (used by arrowHead routine)
    subDocField: Document, --is doc for field holding data I point to
    dataDoc: Document --is document for data I point to
    ];
```

```
PtSubformatData: TYPE = RECORD
    [
    display: DisplayRec,
    form: RECORD[formatIndex: FormatIndex]
    ];

--Layouts:

--Goes in the subformat subdata slot

pLaySubformatData: TYPE = POINTER TO LaySubformatData;

LaySubformatData: TYPE = RECORD
    [
    needed: Rectangle,  --area needed for entire layout
    displayContext: POINTER TO CGraphicsDefs.DisplayContext,
    fields: DESCRIPTOR FOR ARRAY OF LayDocFieldData
    ];

LayDocFieldData: TYPE = RECORD
    [
    ptrID: PtrID,  --used to associate pointer with a layout field
    document: Document,    --is a Layout Field Doc
    formatIndex: FormatIndex      --which format to use for layout field doc
    --unlike everything else, layout fields decide where to put themselves so no Rect here.
    ];

--Fields of layouts:

--For the data slot of formatSet

pLayFieldData: TYPE = POINTER TO LayFieldData;

LayFieldData: TYPE = RECORD
    [
    datatype: TypeID,  --of thing to be displayed in this field
    dataAddr: MemoryAddress  --ditto
    ];

--Goes in the subformat subdata slot

pLayFieldSubformatData: TYPE = POINTER TO LayFieldSubformatData;

LayFieldSubformatData: TYPE = RECORD
    [
    valueDoc: Document,    --document to display value inside field
    formatIndex: FormatIndex,    --format for valueDoc
    valueRect: Rectangle,        --where value will go
    fieldRect: Rectangle      --where field will go
    ];
```

PtData, which contains global information needed by all subformats for pointers, has the following fields:

subDataType which is the type of the data pointed to;

subDataAddr which is the address of the data pointed to. This changes whenever the value in the pointer is changed, whereas the type stays the same; and

ptrID which is a unique identifier used to associate the pointers with the layout field in which to display the referent (see below).

The **PtSubformatData** is divided into two parts, the **form** and the **display**. The form contains the information about how the pointer is to look. In particular, it has a **formatIndex** for the layout field. The **display** contains information about the current display. In particular,

splinePointsUsed specifies whether a spline was drawn for the pointer or not. For pointers that are NIL, for example, a diagonal line is used instead of a spline (see Figure 7.1);
splinePoints contains the sevenPoints used in drawing the spline for the arrow. This allows it to be erased;
destHeight is the height of the side to which the arrow was drawn. This is used to decide how big the arrowhead should be so it must be saved to allow correct erasure;
dataDoc is the document for the data referred to by the pointer; and
subDocField is the layout field document that displays **dataDoc**.

Layouts do not need any global data and consequently only have **LaySubformatData**. The fields of this are:

needed which is the rectangle required by the layout;
displayContext which is the display context used to display the layout itself (see section 7.2.3.2); and
fields which is an array of all the fields of the layout, each of which has a **document** which is the layout field document, a **formatIndex** for that field, and a **ptrID** that associates the field with a pointer.

Layout fields have global data like that for pointers:

dataType and **dataAddr** which describe the type and address of the data to be displayed in them. Layout fields need this information to create a document for their **valueDoc** if it does not already exist (see section 7.2.3.2).

In addition, layout fields have **LayFieldSubformatData** containing:

valueDoc which is the document to be displayed in this field;
formatIndex which is to be used for **valueDoc**;
valueRect which is the rectangle for **valueDoc**; and
fieldRect which is the rectangle for the field itself.

7.2.3.2 Operation of the subformats.

When a layout **subformatProc** is called, it is given a **maxArea** rectangle like all other documents. It then sets the scaling factor so that the **needed** rectangle exactly fits into the **maxArea** rectangle. It is assumed that the proportions of the layout **needed** rectangle were not significant (in fact, the default layouts all use a 100 by 100 rectangle). The current display context is then stored in the layout **subData**. Finally, the layout iterates through the layout field documents causing each to be displayed if it has not been displayed already.

The layout field subformat procedure first tests to see if **valueDoc** is non-NIL. If so, the display context of the layout is used to display the **valueDoc**. If not, a new document is created as described below.

Assume a pointer is enclosed in a record which is inside the first layout field (see Figure 7.8). The record will be told to display by the field and the pointer subformatProc will be called in turn. The pointer document first gets a new memoryAddress for the referent (since it may change). Next EnumerateRegDocs is used to see if a document is already on the screen which has the same memoryAddress as the referent. This allows structures such as Figure 7.9 to be displayed correctly where multiple pointers refer to the same data item. It has the additional advantage that pointers to subparts of records can be handled correctly (Figure 7.10).

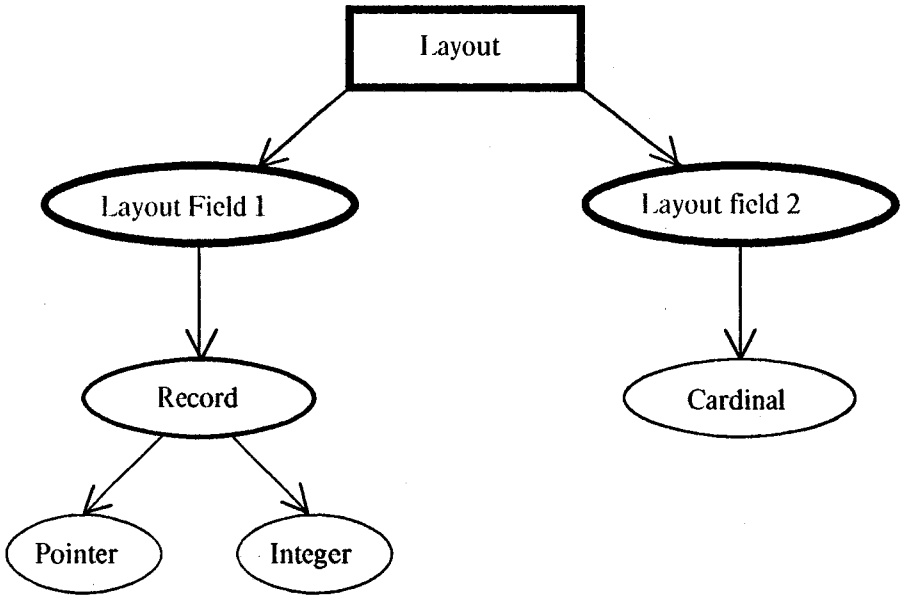


Figure 7.8. Document hierarchy that would be created for:
rec: RECORD [p1: POINTER TO CARDINAL, int: INTEGER];
(This figure was not created by Incense).

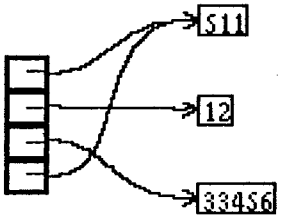


Figure 7.9. Array of pointers with two pointers referring to same value.

If the target document is not already displayed, the pointer will attempt display it. This is handled in the following manner. First, the procedure *FindField* is called using the pointer's parent and the type and address of the data referred to. This procedure traces up the document tree attempting to find a layout field that has the same **ptrID** as the pointer. A **ptrID** is used, rather than simply the field index, to allow aggregates of aggregates (all containing pointers) to work correctly. For example, in Figure 7.11, a record contains another record and both have pointers in record field number one. A unique naming scheme for the pointers is therefore needed. If the search for a layout field is successful, the layout which found it will return the winning layout field. In addition, however, the layout will store the type and address of the data being pointed to in the **LayData** of the layout field. If the search for the field does not succeed, either a symbol for *illegal* (Figure 7.12a) or *unknown* (Figure 7.12b) is used. *Illegal* means that a document was malformed or an attempt was made to use an unimplemented feature of CedarSymbols. *Unknown* is used if there are simply no layouts that want to handle the pointer's referent, but this should never occur with automatically created documents.

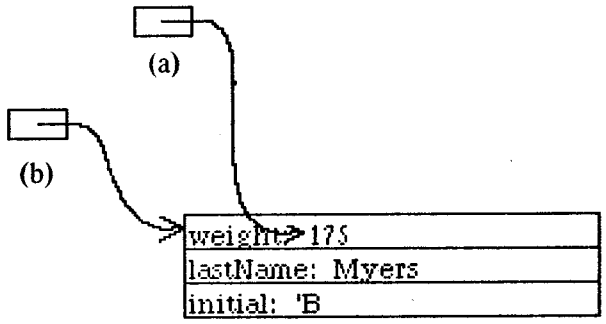


Figure 7.10. Pointer to value inside a record (a) does not get confused with a pointer to the record itself (b).

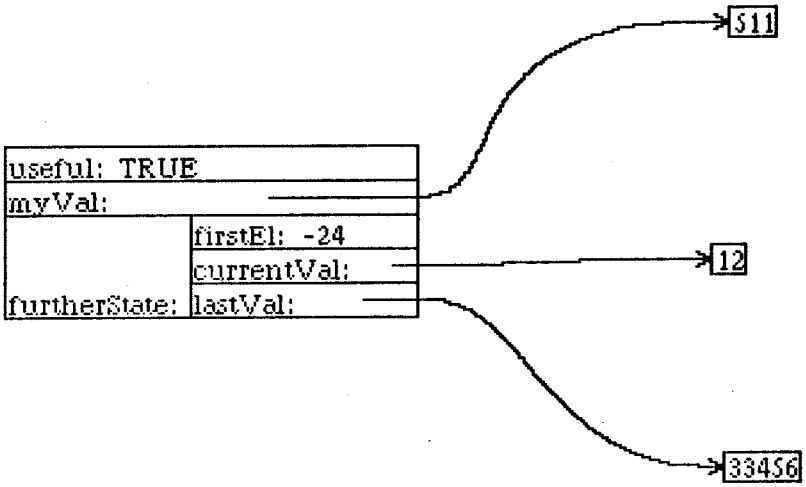


Figure 7.11. Record with internal record both with pointers in record field number one demonstrating necessity for unique PtrIDs.

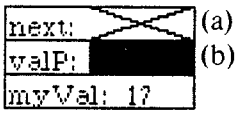


Figure 7.12. Record containing pointers represented as *Illegal* (a) and *Unknown* (b).

Once the pointer finds a layout field, it is displayed. Now control passes to a layout field. As described above, if the field's **valueDoc** is not NIL, the **valueDoc** is displayed in the **valueRect** rectangle. Note that in this case the displayContext from the layout is very important since the layout is *not* causing the layout field to be displayed. Rather, it is the pointer's document that calls the layout field so an arbitrary display context might be current. If the **valueDoc** is NIL, then the field will attempt to create a new document using the data address and type stored earlier in its global data by the layout when FindField was called. It is necessary to be able to create documents dynamically for layouts (unlike for records) because layouts can be used to display structures of indeterminate depth. An example of this is a recursive data type, such as

```
List: TYPE = POINTER TO ListRecord;
ListRecord: TYPE = RECORD [value: CARDINAL, next: List];
```

for which no number of documents can be known to be sufficient *a priori*. The creation process is described in section 7.4. After a **valueDoc** has been created, it can be displayed using the techniques described above.

Returning to the pointer, the field has completed its display, presumably including the display of the document for the data the pointer refers to. Now, however, the pointer must find this document so the arrow can be drawn to it. This is done through the use of the *FindDocUnder* routine in the layout field. This routine matches the memory address and type arguments (which are of pointer's referent) against the address and type of the document and all constituent documents. It is necessary to travel down the document tree because the **valueDoc** in a field may be a layout, and the actual referent could be buried down many levels. Once the **dataDoc** is found, the *DrawArrowFrom* routine in it is called to draw the arrow from the pointer.

7.3 Erasing Documents.

Erasing of documents is not complicated and was essentially covered in section 5.3.3. The current section merely explains the arguments and the actual operation of the erase procedures.

The standard top level erase procedure first checks to see if the document is displayed and if not the procedure simply returns. Otherwise, it de-selects the document if it is selected, de-registers the document, and then calls the internal erase procedure corresponding to the format and subformat used for display (**displayUsed**). After the internal procedure is completed, **displayed** is set to FALSE and the rest of the display information is reset.

The most time-consuming part of the erase is the actual clearing of the screen picture. With records and arrays, some parts of the screen will be erased more than once since both the fields and the aggregate erase the same area. To avoid this, an argument to the erase procedure (*eraseScreen*) determines whether or not the screen area (**myAbsPos**) should be erased. The erase procedures for the non-aggregate documents simply test this boolean and erase if TRUE. Records, arrays, layouts and other aggregate documents, however, first call the erase procedure in each sub-document. In this case, the eraseScreen flag is set to FALSE to save time. Pointers, however, always erase the arrow since it would lie outside of the rectangle of the parent.

7.4 Creation of Documents.

The creation of documents is a relatively complex task, due to the necessity for allocating different types of storage and correctly filling the many fields. Since there is very little consistency checking in documents, it is imperative that the documents be created correctly. This section discusses the generic *CreateDoc* routine and all of the procedures it uses to create documents of the various types. In addition, the creation of documents with client-defined rectangles will be described.

7.4.1 Generic creation routines.

A generic creation procedure, called *CreateDoc*, has been provided. It takes a typeID and a memoryAddress and creates a document using all the defaults. In addition, *CreateDocForVar* is provided as a utility. It creates a document for a variable using its string name and a description of its context. Thus, the simplest way to get the value for a variable myVar using format 0 and rectangle rect is:

```
d: Document ← CreateDocForVar["myVar", ctxTypeID, ctxAddr];
[] ← DisplayDoc[d, 0, rect];
```

A JAM procedure, called *vis* (for "visible") is provided that gets the variable-name string using the keyboard and the rectangle using the mouse and then executes the above code.

CreateDoc is actually a dummy procedure since all it does is call *InternalCreateDoc* with some of the parameters already set. InternalCreateDoc actually does all the work. Its definition is

```
InternalCreateDoc: PROCEDURE [typeID: TypeID, addr: MemoryAddress, layptr, boxed, arrayDown: BOOLEAN]
    RETURNS [d: Document, w, h: Coordinate];
```

and CreateDoc calls InternalCreateDoc with the booleans all TRUE. The w and h returned are the width and height (in a base coordinate system) of the display for the document created. This allows recursive embedding of documents.

First, InternalCreateDoc determines whether a prototype document has been specified for the current display. If so, the prototype is copied into a document which is returned. This test proceeds in three stages, since a prototype can be specified for a *variable*, its *type*, or that type's *base type*. Thus, for example, for

```
Age: TYPE = CARDINAL;
myAge: Age ← 22;
```

the client might specify a prototype for myAge, all data of type Age, or all data of type CARDINAL. If all of these tests fail, InternalCreateDoc does a case select on the base type of the data to find the correct default. The actions for the various types is described in the next sections.

7.4.2 Creation of simple documents.

For data of types UNSPECIFIED, INTEGER, REAL, CARDINAL, BOOLEAN, CHARACTER, WORD, STRING, ENUMERATED, and PROCEDURE no internal data is required. A generic procedure (*TwoSubDoc*) is used to create documents for these types. The procedure is called TwoSubDoc since it fills the document with two subformats (see section 7.2.1). The parameters to this procedure are the two subformatProcs and the address and type of the data. If the *boxed* parameter to InternalCreateDoc is TRUE then the normal and grey boxing subformats are used. Otherwise the non-boxing subformats are chosen. TwoSubDoc allocates storage for a document with one format and two subformats and initializes all the fields. The height returned by CreateDoc is simply the height of the font that will be used to print the the value. The width returned is calculated based on the average or maximum number of characters necessary to write a value of that type. This is multiplied times the average character width. The number of characters used is given in the following table:

Type	TypNumChars	Maximum or Average
UNSPECIFIED	7	max
INTEGER	6	max
REAL	15	ave
CARDINAL	5	max
BOOLEAN	5	max
CHARACTER	2	max
WORD	7	max
STRING	20	ave
ENUMERATED	10	ave
SUBRANGE	TypNumChars[baseType]	---
POINTER	2	---
PROCEDURE	20	ave

Subrange documents require a small amount of extra data, so they are given their own creation routine. Operating similarly to TwoSubDoc, *CreateSubrangeDoc* also allocates and initializes the extra data required.

7.4.3 Creation of record documents.

When InternalCreateDoc is called with a POINTER, RECORD, ARRAY, or DESCRIPTOR FOR ARRAY, it first checks the BOOLEAN argument *layptr*. If this is TRUE and the type contains one or more pointers, then a layout is created instead of the type specified. Thus when a document for an array of pointers is requested, a layout document is actually returned. This allows the pointers to be able to display the referent. Section 7.4.7 discusses the creation of layouts, and this section discusses the case where the record contains no pointers or layptr is FALSE.

The record construction procedure (*Create2SubRecDoc*) takes as arguments the type and address of the record and three procedures. These procedures are used to get the rectangles and arrow end points used in designing the form for the record. In automatically created record documents, the procedure orients the rectangles stacked vertically. The size used for each field is derived from the width and height of the field document. The definitions for record creation are:

```
--Records
Create2SubRecDoc: PUBLIC PROCEDURE [recType: TypeID, recAddr: MemoryAddress, getRects: RecGetRects,
    adjustRects: RecAdjustRects, get3Points: Get3Points] RETURNS [d: Document, w, h: Coordinate];

--the following is used to get or generate the rectangles to be used for the field and value (data) in a record field.
RecGetRects: TYPE = PROCEDURE [fieldName: STRING, fieldType: TypeID, xStart, yStart, valueW, valueH:
    Coordinate] RETURNS [nameR, valueR, sum: Rectangle, nameW: Coordinate];

--The following is called after all fields are processed and can modify the rectangles to put in correct place, make sum
    rectangular, etc. (May be a do-nothing procedure).
RecAdjustRects: TYPE = PROCEDURE [recDoc: Document, which: SubformatIndex];

--The following (called after all fields are adjusted for records) is used to get the 3 points needed to point to this
    document (record or array).
Get3Points: TYPE = PROCEDURE [d: Document, which: SubformatIndex] RETURNS[attach: ThreePoints];
```

The operation of Create2SubRecDoc is as follows: first, the number of fields of the record is discovered using the CedarSymbols command *GetMaxIndex*. A document is then created with one format and two subformats. Next, the RecSubformatData and RecFieldData data structures are allocated. An internal procedure, called *MakeRecField*, is then called to fill in each field. MakeRecField is defined as:

```
MakeRecField: PRIVATE PROCEDURE [i: CARDINAL, recType: TypeID, recAddr: MemoryAddress, psubData:
    pRecSubformatData, getRects: RecGetRects, parent: Document, xStart, yStart: Coordinate] RETURNS [xEnd,
    yEnd: Coordinate];
```

where i is the field index. MakeRecField uses the recType to get the field name, and it creates a document for it after appending ": " to the end of the string. The recType, along with the recAddr, is also used for getting the type and address of the i^{th} component of the record. InternalCreateDoc is then called by MakeRecField to create the document for the field. The arguments are the field's type and address, FALSE for **layptr** and **boxed**, and TRUE for **arrayDown**. Thus, the record field components will not box themselves and they will not create internal layouts since this would have been handled first at the top. The width and height returned from InternalCreateDoc, along with the field name string, the field value type, and xStart and yStart, are all used as arguments to the *getRect* procedure. For the case where the user explicitly specifies the

rectangles (see Figure 5.8), this routine ignores all the arguments. For the automatic case, however, they are used in the following manner: the width (nameW) and height (nameH) of the field name are calculated using CGraphics routines. The rectangles are then calculated as follows:

```
nameR ← [xStart, yStart, xStart + nameW, yStart + nameH];
valueR ← [xStart + nameW, yStart, xStart + nameW + valueW, yStart + valueH];
```

Thus the rectangles are situated in the correct manner with the field name to the left of the value. Note that this recursive creation allows a display of any size to be included as the record value with no special work.

One problem with this scheme, however, is that the right end of the record display would end up very ragged. A *RecAdjustRects* routine is therefore called after all the fields have been created. This iterates through all the field rectangles to find the one with the greatest width. All the other rectangles are then modified so they line up on the right. When the user is specifying the rectangles, RecAdjustRects does nothing. Finally, after all this has been completed, the **needed** rectangle is calculated by summing all of the other rectangles. Using this rectangle, the *Get3Points* routine then calculates the destination points for arrows. The automatic Get3Points simply uses the left edge of the **needed** rectangle for the first X value and the top of the **needed** rectangle minus ½ the height of the current font for the Y value. The other two points are five and ten screen points directly left of the first.

7.4.4 Creation of array documents.

Arrays are created in a manner very similar to records. The major difference is the use of the *arrayDown* argument. This controls the direction the rectangles are stacked for the automatic creation of rectangles for the array elements. Thus, one dimensional arrays have the first element at the top and the last element at the bottom (Figure 7.13a). When the InternalCreateDoc is called for the next level, the arrayDown flag is complemented so the next level arrays will be arranged with the first element on the left and the last on the right (Figure 7.13b). Thus, a two dimensional array will be displayed in two dimensions (Figure 7.14a). For three dimensional arrays, the the third dimension will fit vertically into the rectangle for the the second dimension (Figure 7.14b). The array creation procedure (*CreateArDoc*) does not need an AdjustRects procedure since all the elements are the same size (being the same type).

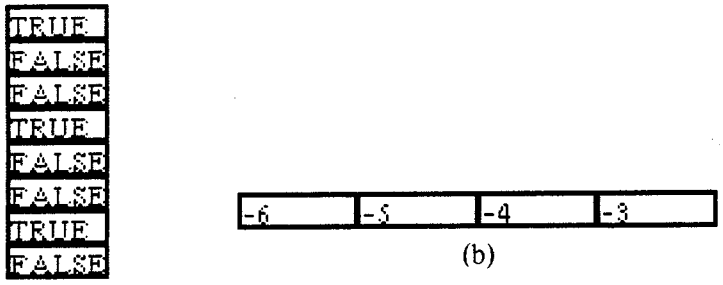


Figure 7.13. Arrays oriented vertically (a) and horizontally (b).

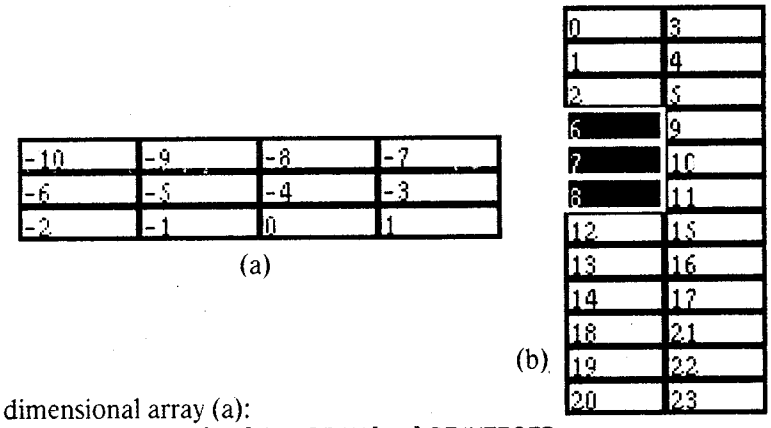


Figure 7.14. Two dimensional array (a):
ARRAY [1..3] OF ARRAY [1..4] OF INTEGER;
and three dimensional array (b) with one internal array selected:
ARRAY [1..4] OF ARRAY [1..2] OF ARRAY [1..3] OF CARDINAL;

7.4.5 Creation of pointer documents.

Once the layout has been created, a pointer document is still needed to draw the actual arrow. The complexity is all actually in the layout and the pointer procedures; the pointer document itself is very simple. The only item of interest is the *PtrID* which is associated with the pointer. This is generated by a simple procedure that ensures that there will never be two ptrIDs alike (see section 7.2.3.2). The constituent documents and address of the referent are all set to NIL so they will be created at run-time.

7.4.6 Creation of array descriptor documents.

Array descriptors are used primarily to implement variable size arrays. They are composed of a length and a pointer to an array. Therefore, the display for them is simply a CARDINAL and a pointer (Figure 7.15). This is handled in a manner very similar to records and pointers. A document is created for the descriptor and then one each for the length and pointer. The length document is the same as other CARDINAL documents, but the pointer requires a special subformatProc, however, since it needs to get the address of the referent (the array) from the

descriptor and not the *pointer.* This is a minor complication, however, and the rest of the operation is the same.

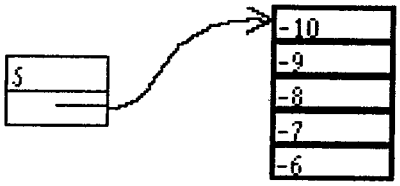


Figure 7.15. Display for a DESCRIPTOR FOR ARRAY [1..5] OF INTEGER;

7.4.7 Creation of layout documents.

Layout documents are constructed in a similar manner to record documents. The *CreateLayout* procedure has the following definition:

```
CreateLayout: PUBLIC PROCEDURE [typeID: TypeID, numFields: CARDINAL, addr: MemoryAddress, getRects:
    LayGetRects] RETURNS [d: Document, w, h: Coordinate];
```

The number of fields for the layout is calculated by the InternalCreateDoc by adding one to the number of pointers in the data structure to be displayed. The procedure which counts the number of pointers is recursive since it must include in the count pointers in all internal data structures (such as records in records: Figure 7.11). The document for the data structure will be placed in layout field zero. The operation of CreateLayout is as follows: first a document is allocated for the layout, and then InternalCreateDoc is called for the data structure for field zero. The arguments are the typeID and address passed to CreateLayout, layptr: FALSE, boxed: TRUE, and arrayDown: TRUE. The next step is to get all of the PtrIDs used in that document. These will be distributed among the fields of the layout. Finally, a layout field document is created for each field and the rectangles for it are found using the *layGetRects* procedure. Documents for the value in each field are *not* created here, however, since they will be created at run-time.

The layout routine for automatically constructing the rectangles for the fields works somewhat differently than that for records or arrays. Its definition is:

```
LayGetRects: TYPE = PROCEDURE [i: FieldIndex, r: Rectangle, f0w, f0h, inc: Coordinate] RETURNS [fieldR, valueR:
    Rectangle];
```

where i is the the index of the field, and r is the rectangle for the entire layout. Note that this is different from the record case where the size of the whole record was the sum of the constituents. Here, the constituents are expected to get their rectangles by partitioning the larger rectangle. This is done using the width (f0w) and height (f0h) of field zero and the height of each of the fields (inc). When the client is defining the rectangles, he can simply divide the rectangle in any manner desired. The automatic generation routine gives one-third of the area to the first field and divides the rest vertically for the rest of the fields. Figure 7.16 shows the rectangles for a layout with 4 fields. For field zero, the value is given slightly less room in the horizontal direction than the field to ensure that there is sufficient room for the arrows to exit the pointers on the right (Figure 5.6). f0w and f0h are not used in the current algorithm which is:

```
IF i = 0 THEN
        BEGIN
        valueR ← [r.lowerX, r.lowerY, r.lowerX + (2/9)*rWidth, r.upperY];
        fieldR ← [r.lowerX, r.lowerY, r.lowerX + (1/3)*rWidth, r.upperY];
        END
ELSE    BEGIN
        valueR ← [r.lowerX + (1/3)*rWidth, r.lowerY + (i-1)*inc, r.upperX, r.lowerY + i*inc];
        fieldR ← valueR;
        END;
```

The constituent documents of layouts are thus given an amount of room for display which is entirely independent of the amount of room they desire. This was the motivation for having records, arrays, etc. center themselves in the vertical direction.

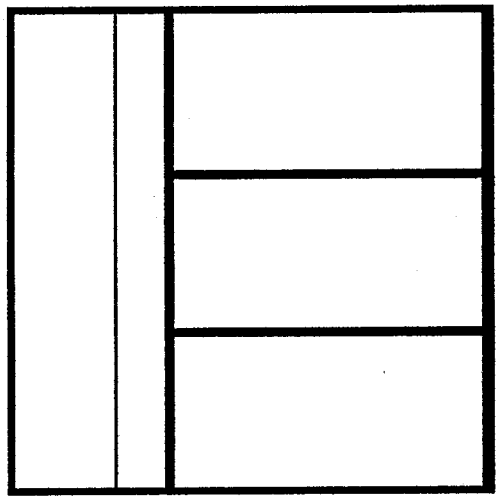


Figure 7.16. Rectangles for fields and values in a layout with 4 fields. The bold lines are the field rectangles, and the rectangle to the left of the thin line is a value rectangle for field 0. The other value rectangles equal the field rectangles.

7.5 Client-defined Documents.

One of the chief goals of Incense was to allow the client to define, create and use his own documents. This section will investigate how this can be done and the facilities available for aiding the process.

Incense allows the user to define his own global and subformat data and write his own procedures for displaying and modifying documents. This provides a structured environment in which the client can specify his own document displays. The utility procedures provided by Incense and CGraphics make many types of displays very simple. Thus, for example, the "percent-done thermometer" of Figure 2.1 could be implemented by writing only one small procedure for the subformatProc since procedures already exist that are appropriate for the rest of the operations.

One problem that might be encountered when creating documents is that it is very difficult to add procedures to the previously defined documents. Layouts, for example, required the

DrawArrowFrom, *FindField*, and *FindDocUnder* procedures to be added to all documents. To add another document type like layouts that required *all* documents to have new procedures would require recompilation of all of Incense and some other major changes. This problem, unfortunately, arises directly from the structure of Mesa and would be hard to program around.

Once a document prototype is created, the client can then simply associate it with a variable or type and have it used whenever appropriate. The original default display can be included as one format of his document. Then the client can choose at run-time in which format the data structure will be displayed. The clock document was implemented in this manner.

VIII. Ideas for Future Work

Although Incense is a powerful system and has the capability to display data of almost every type in Mesa, there are many dimensions in which it could be extended. These can be divided into four classes: improvements to the current Incense prototype, modifications to make Incense a production system under Cedar, special purpose documents for specific types, and changes that would require major alterations to the Incense system.

8.1 Improvements to the Incense Prototype.

Incense and CedarSymbols currently are not complete systems. A number of planned and desired features were not implemented due to the lack of time. This section describes these features, which involve little or no change to the basic structure of Incense.

8.1.1 CedarSymbols.

Much of the code used in the current prototype implementation of CedarSymbols was copied from the current Mesa compiler, where the code is used for debugging the compiler itself. Ed Satterthwaite was invaluable in explaining what was going on and how to access and interpret the exceptionally complex symbol tables produced by the compiler. Largely due to time limitations and this complexity, which stems from the attempt to minimize the size of the tables, some of the functions described in chapter 6 have not been implemented.

The most serious limitation is that only global variables in modules can be accessed. In addition, there must be enough memory available to hold the entire symbol table for that module. This symbol table must be kept in memory while any of the typeIDs based on the associated context are in use. In a real implementation, a more clever storage management scheme would be needed since symbol tables tend to be very large.

In addition, variant records and the subcomponent operations on transfer types, processes, conditions, and certain types of pointers are not implemented. Also, memoryAddresses can only be created for constants that take one word that contain POINTERS. In addition, due to the lack of a garbage collector, the client of CedarSymbols must deallocate all typeIDs and memoryAddresses explicitly. These repairs would require a fairly large amount of work by an experienced Mesa systems programmer.

In order to be able to associate a prototype with a particular type, Incense needs to be able to find out whether types are equivalent. CedarSymbols currently allocates a new TypeID every time one is returned. A procedure is therefore needed to determine if two typeIDs correspond to the same type. In addition, it would be useful to have a unique typeAtom for a typeID that could be used to test equality.

8.1.2 Prototype Documents.

Prototype documents currently are not implemented. They would require the type equivalence tests described above and a method for copying the relevant parts of a prototype document. A special prototype document type that understands the message *Copy* may provide the latter part of feature. A list of prototypes and the types they go with would also have to be maintained by Incense.

8.1.3 Editing.

The edit message is not implemented for any documents. Part of the problem is deciding where to get the new value. The current design has the edit procedure take a memoryAddress for the new value. This does not allow special types such as pointers to accept their values by pointing, etc. A final design for the edit command must await the Cedar specifications of user input.

8.1.4 Additions to arrow display.

At present, documents do not remember anything about the arrows that were drawn *to* them. Thus when a document is moved, the arrows that previously pointed to it are left dangling. Since the destination documents are responsible for drawing the arrows from the pointer documents, information could easily be stored that would allow the document to cause redisplay of all arrows to it if it moved.

Another small and useful modification to arrows would be to allow the user to specify a new point (*knot*) for the arrow to go through. This would allow the user to move a line out of the way of other objects on the display. The only trick to implementing this modification is deciding the two knots in between which the new point goes.

8.1.5 Creating a forms editor.

Currently, it is very difficult to specify the form for a document. (The *form* contains the various rectangles and connection points.) A display-based graphical tool for creating and editing the pictures would therefore be very useful. It might allow the user to draw icons to be used as subformat displays of documents. A forms editor would actually be a another piece of a debugging system and should not require any modifications to Incense itself.

8.2 Making Incense a Production System.

Incense will eventually be converted into a production system as part of the Cedar debugger. For this to happen, however, a large number of modifications and enhancements need to take place. Many of these stem from the additional facilities that will be provided by Cedar, but others will be performance tuning.

8.2.1 Utilizing Cedar language features.

Cedar Mesa will contain a number of useful features that Incense has been designed to exploit. A run-time type system that will tie into CedarSymbols should allow Documents to be entirely type-safe. Thus all the LOOPHOLEs and POINTER TO UNSPECIFIEDs could be eliminated. In addition, this will allow garbage collection of documents and their constituents. The language is also supposed to be extended to promote object-oriented programming. This will clearly be beneficial and useful for Incense.

8.2.2 Utilizing Cedar documents.

Apart from the language changes, Cedar will include sophisticated user-interface mechanisms for input and output. The notion of a *document* in Cedar has taken on additional structure since the Incense documents were designed and these changes should be incorporated. This will allow Incense's displays to be shown in a window while other activity occurs in other windows similar to the way operations are handled in DLISP (Figure 3.5) and Smalltalk (Figure 3.3). In order to make this work properly, documents in Incense will need better control over their display. Documents in Cedar are allowed to be displayed an arbitrary number of times simultaneously in different styles. This will be confusing in Incense, however, because the destination for arrows will not be well-defined.

8.2.3 Adding *Views* to Incense.

The notion of a *view* has been developed to take care of this problem. Each Incense display will take place in a view, which might be a rectangular window or may simply be a logical organization. The restriction will be that there can only be one document associated with any data structure in a view and that document can only be displayed there once. Thus, if the user selected a document and requested that it be redisplayed in the same view as the original, the original would have to be erased. If the document were to be redisplayed in another view, the original could be left alone. The additional restriction that makes this all attractive is that arrows never cross from one view to another. Views could be added to Incense with a minimal amount of additional mechanism. The registering facility would be extended to include the view, and the creation routines would take a view as a parameter and check to see if a document for the data was already there.

8.2.4 Increasing the speed of Incense.

Incense will execute faster under Cedar since it will be run on faster machines. In addition, a general cache mechanism is being designed for Cedar that should allow bitmaps and splines to be stored to allow much faster redisplay. If some ability to define events exists (as in section 3.1.4),

monitoring of the program's activity using pictures will become feasible. Presently, the Incense display on an Alto is too slow to be reasonably used in monitoring.

Other speed improvements can be achieved by optimizing and simplifying some of the complex mechanism used for displaying layouts. Also, some of the internal data structures, such as the list of all documents displayed, could be profitably changed to hash tables.

8.2.5 Using Cedar's history facility.

Plans for Cedar include a sophisticated history facility. Presumably, it will allow saving of arbitrary events that could be used to enact a replay. Incense could be modified to store in this history the this information. Thus, the advantages of real-time monitoring and replays using analogical displays could be achieved using Incense.

8.2.6 Removing Incense from JAM.

Clearly, Incense needs a more powerful user interface than JAM. Once a debugging system for Cedar is developed, Incense should be integrated with it. This should also solve the problems with editing and accepting other user commands. The programmer should be given the option of using Incense if desired, but other methods of data display will probably be available. Further, the debugger should allow specification of documents for data structures at run-time without destroying the state of the program being debugged.

8.3 Special Purpose Documents for Specific Types.

Once documents can be associated with specific types, there will be a great temptation to develop prototype documents that use more sophisticated displays for various types. In addition to the display, however, some thought must be given to how the programmer will be able to edit the values through these displays (since conversion may be necessary). Another problem is what to do with illegal values. A library of successful prototype documents might be maintained so that anyone could use them. Some ideas for such documents are listed in this section and most could be implemented either in current or Cedar Incense.

Percent-done thermometer. This would be useful for showing progress in loops or for variables representing percentages. The only thing special about this document is that the maximum legal value would have to be stored to allow calculation of the correct percentage.

Progress in program as arrow to source. This would be slightly harder since the conversion would have to be made between the program counter and the actual source statements. This is clearly

possible since the debugger does it for breakpoint setting, but requires added sophistication in CedarSymbols.

Mesa run-time stack display. A more likely (and possibly more useful) facility than showing the progress in the program would be a symbolic representation of the run-time stack of Mesa. This also requires new facilities from CedarSymbols. The current Mesa debugger is particularly deficient at handling multiple processes, but if Incense could graphically display the stacks from all current processes, the user would be able to move around much more easily. The complex context mechanisms of the current debugger and CedarSymbols could be hidden by allowing the user to simply select a stack frame using the mouse. A more generic stack prototype document might also be useful for displaying stacks that the programmer created.

Index into an array as pointer. This would be simple to implement using the tools and procedures already available.

Time and Date as string. The time and date can be most concisely represented as the string: "4:32 PM, 12/15/79". This is an example of a large class of types where a larger structure (such as a record) is used to represent a simple concept.

Arrays and Records as pretty-printed string. Sometimes the user may not feel that the graphic display gives him any insight or that it is too costly in terms of execution time or screen space required. In that case, a pretty-printed display of a textual representation might be more appropriate.

Ring buffers, lists, and trees. These are example of common data structures used in programs that would probably be easier to debug and understand with special displays. List and trees are sufficiently well structured that much better space allocation can be done than is possible for layouts which must handle arbitrary structures. In addition, if these structures are used to represent a variable length array or other higher-level abstraction, all of the pointers might be omitted in favor of the array or some other notation. A ring buffer might be represented as a circle with the appropriate parts marked as in Figure 8.1.

Pie charts and bar graphs. These analogical displays would be very simple with the tools already provided by Incense and CGraphics. A small amount of extra data would have to be stored to describe the maximum or total values so the correct distances could be calculated.

Single variable trace of values. EXDAMS (section 3.5.3.1) allowed the user to request a "flowback" trace of how a variable received its current value. If the history facility of Cedar were sufficiently powerful, Incense could offer the same capability. Even if it were not, a variable could be displayed as a special type of array representing all past values. This would only require some way of monitoring variables to collect the values as they were assigned.

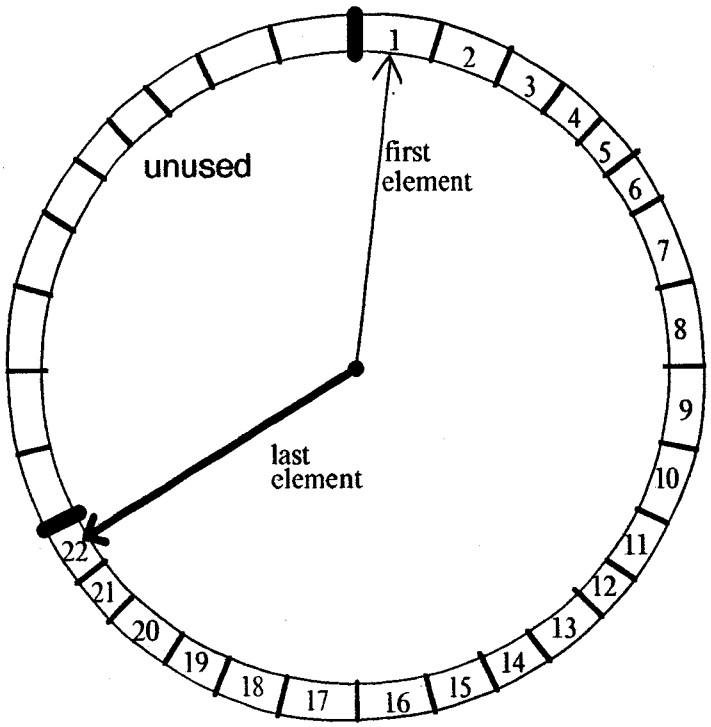


Figure 8.1. Possible display for a ring buffer.

Better display for arrays. There are many ideas for improving the rendition of large structures such as arrays. For example, a small portion of the array might be presented in a window and the user could scroll up and down to find the interesting values. Also, some means of representing the indices is needed.

Array as bitmaps. Sometimes arrays are used to hold actual pictures. For example, the cursors shown in Figure 4.3 were stored as arrays of octal numbers. Some method of converting these into the actual pictures would be useful.

8.4 Improvements That Require Major Alterations.

In addition to the ideas presented above, there are others that would require a major amount of work, or which are even beyond the present state of the art. Just the same, they might be useful in a future system if an implementation could be achieved.

8.4.1 Unifying typeIDs and memoryAddress.

One problem with CedarSymbols is that the client must be careful to ensure that the typeIDs and memoryAddresses are consistent. It is always an error to use a memoryAddress and typeID

together that do not correspond to the same data structure. Therefore, these concepts could be combined to make the system easier to use. In addition, a fully object-oriented approach could be adopted to allow a type-safe way of getting and changing values with only a small decrease in efficiency.

8.4.2 General two-pass display.

Currently, Incense uses a simple one pass algorithm to display documents. The location of all documents on the screen is rigidly set before the document is displayed. There is no notion of the document negotiating with the parent for "just a little more room." It should be clear from the text and pictures in this paper that this does not achieve maximal utilization of the screen space in some cases. Another method for displaying would be to have the parent ask the subordinate document how much room it would like and then calculate the room to be used by other documents based on how much room was left. This requires a fairly detailed two-dimensional "free-space" map to be kept from which documents could use and return screen space. The reason the tree and list displays could use the space so much more efficiently than layouts is that they could count on allocating space exactly along a single line since the size of all nodes is known. This one-dimensional space allocation problem is clearly much easier to solve.

An additional problem with two-pass space allocation is that the documents may do as much work deciding how much room they need as they would have to actually display it. Some caching mechanism is therefore called for. If the parent decides the subordinate cannot have as much room as it requested, however, this work may be entirely wasted. Thus the two-pass scheme has the potential to be very unstructured and possibly more complex, slow and expensive than warranted for a debugger's data display.

8.4.3 Remote monitoring.

One of the major applications for the Alto and PARC's other research computers has been the development of screen-oriented systems. For example, there is a menu-driven multi-window mail program, a full-screen editor, etc. Monitoring of these programs is therefore difficult since the screen is completely filled by the application programs. Flashing back and forth from one environment to another would be very disturbing and probably not very enlightening. In addition, some of the applications are computation-intensive and monitoring tools that substantially slowed them down would not be acceptable.

All of PARC's computers are connected to each other using the Ethernet local communications network [Metcalfe 76] so it seems natural to try to take advantage of this for monitoring. If a facility for making a history were available, it should be a natural extension to put all this information into a packet as it is gathered and then send it over the Ethernet to another Alto. The

user could then monitor his program remotely. There might even be remote facilities for interrupting and interrogating the program while it is executing. Model's system (section 3.5.3.6) allowed a similar form of remote monitoring.

8.4.4 Ideas from artificial intelligence and program methodology.

Some features that might be added to Incense must await further advances in artificial intelligence and other fields of computer science before they can be practical. Such things as a natural language front end and the ability of the system to answer questions *about* the data would clearly be useful. In addition, work in program methodology may eventually progress far enough so that Incense could generate a picture of a "typical" data structure instance from only the type. This would be useful for documentation so the user would not have to explicitly create the display, as was done, for example, in making Figure 7.1.

IX. Summary and Conclusion

This thesis describes a system called *Incense* which graphically displays data structures of various types for the language Mesa. Currently running on an Alto mini-computer, Incense can display all of the basic types of Mesa in a reasonable textual form. It uses a more analogical format for the aggregate types such as arrays and records. A major advantage of Incense over other data display systems is its ability to display pointers as arrows to the actual target data. The displays are automatically created for variables with minimal interaction required of the user, yet a programmer can easily create his own displays if desired. A structure has been provided for storing these prototype displays in a library and associating them with specific types or variables. Finally, the client can specify a number of different formats and subformats that allow the display of a particular data structure to be changed by the user or program based on various criteria. Thus, a display can be automatically generated that is very similar to the picture the programmer might have drawn if he were explaining his data structure to another programmer.

Incense has solved a number of difficult problems to achieve this level of performance. *Layouts* were developed to avoid the expensive and difficult two-dimensional space allocation problem and to allow one-pass display for all data types. Layouts also allow the client to specify where the referents of pointers should be drawn. In addition, an abstract interface to the Mesa symbol tables was defined that successfully allows Incense to avoid knowing any details of the implementation of Mesa types.

The design and implementation of Incense has incorporated many of the appropriate features of a debugging system as discussed in chapter 2 and section 5.2:

- **Reduce the volume of data required from the user**: If the user requests a display for a pointer, the referent is automatically displayed also. Selections and many commands can be done using the mouse.
- **Adjust the form of the input to make good use of the human faculties**: The user can point at and move the displays using an analog device (the mouse and its buttons).
- **Use feedback to allow correction of mistakes as they are made**: The selected document is video-reversed so the user can tell what is selected. Boxes are drawn when the user specifies a rectangle for a document.
- **Adjust output quantity to human capacity**: The display may fill up but the user can concentrate only on those portions that interest him. When many things are displayed, they decrease in size or disappear so that the screen is not cluttered.
- **Choose forms of output that are readily acceptable to human comprehension**: This is one of the main objectives of Incense. Incense does promote analogical pictures and automatically produces organized displays for data structures.

- **Output only completely processed results:** Incense allows the user to develop documents for any data type. This allows presentation of data structures at the highest conceptual level appropriate. All of the pre-defined Mesa types currently are presented completely processed.
- **Appropriate level of detail:** Incense allows the user to specify that certain fields of a record should not be shown. He can also specify an iconic format for some data structures or fields to reduce the amount of information presented.
- **Automatically generated pictures.** Incense creates the pictures based on the type of the data structures without requiring user intervention. A picture can be produced simply by specifying the variable to be looked at and the rectangle into which to fit the display. A hardcopy can also be easily made of the displays (as was done for this paper).
- **Replay:** Replay is currently not possible, but should be in Cedar Incense.
- **Easy to use:** Incense allows the use of the mouse for selection and specifying rectangles. The other issues of user input will be handled by the system into which Incense is incorporated.
- **No modification of source program:** The source program does not need to be modified to use Incense, although an additional program might have to be written to specify any special documents required.
- **Extensible:** The user can create new documents and associate them with particular types. Libraries of documents could be maintained. In addition, a document for a data structure might allow it to be displayed in a number of formats.
- **Fast:** Incense currently actually allows the investigation of certain data structures at a rate faster than the current Mesa debugger. This is probably an unfair comparison, however, and Incense clearly needs to execute faster. The Cedar implementation will run at least three to ten times faster due to some reprogramming and better hardware and microcode support.

The displays produced by Incense will be useful to programmers. Sutherland [63, p. 67] claimed that "it is worthwhile to make a drawing on the computer only if you get something more out of the drawing than just a drawing." The pictures and the associated data structures can be dynamically rearranged and modified. The displays from future Incense systems will be useful for monitoring of running programs. Since the user can specify documents, the resulting pictures can be used to provide documentation for the data structures themselves (as in Figure 7.1 and 7.2). Finally, debugging will probably be more fun when using pictures rather than long strings of characters. This combined with the higher conceptual level provided by the pictures may make the debugging task easier and thereby increase programmer productivity and reduce the number of missed bugs.

Thus, while it could not actually be described as *finished*, Incense has fulfilled most of its goals. It was an enjoyable project designing and implementing Incense, and I appreciate the opportunity to be a part of the Cedar effort. It should be exciting to fit Incense into Cedar and to study its actual use in the debugging of programs. Only this final test will demonstrate if analogical display of data structures as provided by Incense increases programmer productivity.

APPENDIX A. Informal Poll on the Current Mesa Debugger

A.1 The Questionnaire.

As part of my research for the discussion of section 4.2.1.2 on the current Mesa debugger, I distributed a questionnaire about the debugger. The questions were:

CONTEXT:

(1) Compare the Mesa debugger to others you have used and list the advantages/disadvantages.

(2) Can you debug faster with the Mesa system than with your previous system(s)? Why?

GENERAL:

(3) What do you like about the debugger?

(4) What are the debugger's biggest weaknesses?

(5) Where does the debugger fail to allow you to "think in Mesa"?

DATA DISPLAY:

(6) How would you like data structures (records, pointers, arrays, etc.) to be displayed and would that help you debug programs?

(7) What are the hardest problems to debug and could a clever data display system help with them?

(8) How much additional delay would you be willing to tolerate to have your data structures displayed as pictures?

(9) Would you ever be willing to write a program to specify the picture to be used for displaying data of a certain type? If so, for what kinds of data types?

A.2 Results.

I received 19 responses from Mesa users. They were generally not surprising and are summarized below:

Most people answered questions 1, 3 and 4 together. The advantages of the Mesa debugger were given as:

- The debugger's knowledge of user-defined types and local variables;
- The ability to display multi-item data structures symbolically all at once;
- The ability to avoid knowledge of machine instructions.
- The ability to set an apparently arbitrary number of breakpoints by pointing at source text;
- The ease of using the debugger for beginners. (Note: This was disputed by other respondents);

- The existence of an interpreter;
- The ability to look at source code while debugging; and
- The lack of planning required to use the debugger.

Disadvantages of the Mesa debugger were given as:

- The slowness of many operations such as getting into the debugger and finding the types of data when using multiple symbol tables;
- The debugger's lack of the ability to mix and manipulate contexts;
- The inability to patch code and proceed from errors. In fact, a Mesa program cannot be continued in any location except exactly where it stopped;
- The incomplete interpretive mode: many of the types and operations of full Mesa are not supported in the debugger's interpreter;
- The lack of conditional breakpoints;
- The lack of the ability to automatically display certain variables and then proceed at a breakpoint (which would allow easier monitoring);
- The lack of the ability to omit from the display selected portions of arrays or records;
- The lack of the ability to use command files with the debugger;
- The reliability problems with the debugger itself: "A debugger must be trustworthy;"
- The lack of the ability to monitor variables by having them continuously updated on the screen;
- The extreme difficulty of single-stepping through a Mesa program;
- The lack of information provided about a signal or error if the symbol table describing it is not on the disk. (All that is displayed is an octal number);
- The inability of the user to provide formatting information and/or validity/consistency checks;
- The excessively symbolic and verbose command language of the debugger; and
- The inadequate presentation for arrays.

In answer to question 2, almost everyone agreed that he could not debug faster in Mesa than in other languages. Those who disagreed did so only because the Mesa compiler caught many bugs that would have gotten through in the other languages. In addition, some felt that the Mesa debugger allowed them to debug faster when dealing with highly typed data structures since it interprets the data. Everyone felt that the debugger was too slow, however.

Although the debugger attempts to prevent the user from having to use the underlying representations for data and instructions, it does not always succeed in allowing the programmer to "think in Mesa" (question 5). Some places where this happens are:

- When using monitors;
- When using COMPUTED VARIANT RECORDS or OVERLAID VARIANT RECORDS;
- When using enumerated types in the interpreter;
- When investigating variables in catch phrases for signals;
- When investigating multiple processes; and
- When trying to construct arguments for calling of procedures from the debugger.

On the data display part of the questionnaire, the responses were encouraging for Incense. For question 6, some people felt that the "facilities of Incense are in the right direction" but should be augmented with some ability to "window" onto large structures. Most people felt that some method for pretty-printing of aggregate structures was necessary, even if only textually. Since one can split and scroll the debugger's window, many felt there was no reason not to use extra spaces, tabs and carriage-returns to make data structures easier to read. Others disagreed, however, and thought it was more important to have a lot of information on the screen at once. Some specific data types were mentioned as problems: one respondent wanted to be able to use the mouse to move around in a data structure tree, and another mentioned that it would be nice to be able to display sparse structures such as hash tables in a reasonable manner.

There was surprising agreement in answer to question 7 concerning the hardest problems to debug. The two problems mentioned most were random core smashes and timing problems related to multiple processes. It was felt that a data display system would not help with this, unless it allowed true monitoring of variables and control flow. Other useful features would be user-defined write protection and "a flavor of breakpoint that invoked frequent consistency checking on the data structures involved." One person mentioned that a hard problem is ensuring that all storage is deallocated when no longer needed.

For the delay that people would tolerate (question 8), most people said "none". A few said that they would tolerate a small amount if the overall time it took to display a data structure was faster than without the pictorial display, and one mentioned that as long as he could watch it happening, up to 10 seconds to complete the drawing would be tolerable. If the fancy display took longer, it should be an option and not the default, according to one respondent.

The responses to question 9 were also very encouraging. Most people felt that they were currently in the habit of writing data display routines to allow debugging of complex structures. Other people claimed they might do this if there were fancy tools to encourage it and it could be done in a "simple language." Otherwise, the displaying procedures would have to be debugged and "debugging the debugging facilities is almost always a waste of time." Many people hoped that a large number of standard displays would be available in a library that they could use or modify slightly to display their own data structures.

BIBLIOGRAPHY

[Aaronson 61] David A. Aaronson and Clarissa J. Kinnaman. Production of Large and Variable Size Logic Block Diagrams on a High Speed Digital Computer. AIEE Paper CP 61-1116, Oct., 1961.

[Ahlberg 67] J. H. Ahlberg, E. N. Nilson, and J. L. Walsh. The Theory of Splines and their Applications. New York: Academic Press, 1967.

[Aho 78] Alfred V. Aho and Jeffery D. Ullman. Principles of Compiler Design, Reading, MA: Addison-Wesley Publishing Company, 1978.

[Atwood 78] Michael Atwood and H. Rudy Ramsey. Cognitive Structures in the Comprehension and Memory of Computer Programs: An investigation of Computer Program Debugging. Englewood, Colo: Science Applications Inc. Tech Report TR-78-A21, Aug, 1978. 80 pages.

[Baecker 76] Ron Baecker. "A Conversational Extensible System for the Animation of Shaded Images," Proceedings of ACM SIGGRAPH Symposium. Philadelphia, PA, June 1976.

[Balzer 69] R. M. Balzer, "EXDAMS -- EXtendable Debugging and Monitoring System", Proceedings AFIPS Spring Joint Computer Conference. 34, 1969. pp 567-580.

[Bell 71] C. Gordon Bell and Allen Newell. Computer Structures: Readings and Examples. New York: McGraw-Hill Book Company, 1971. pp. 37-45, 139-144.

[Blair 71] James Curtis Blair. An Extendible Interactive Debugging System, Purdue University PhD Thesis, June, 1971. Ann Arbor, Mich: University Microfilms International, 1979. 151 pages.

[Bobrow 77] Daniel G. Bobrow and Terry Winograd. "An Overview of KRL, A Knowledge Representation Language," Cognitive Science, Vol. 1, No. 1, Jan, 1977. Also available as Palo Alto: Xerox PARC CSL-76-4, July 4, 1976.

[Boehm 71] B. W. Boehm, M. J. Seven, and R. A. Watson. "Interactive problem-solving--An experimental study of 'lockout' effects," Proc AFIPS 1971 Spring Joint Computer Conference. AFIPS Press, Montvale, NJ. pp. 205-210.

[Borning 79] Alan Borning, ThingLab--A Constraint-Oriented Simulation Laboratory, Stanford Unviersity Department of Computer Science PhD Thesis STAN-CS-79-746. Also available as Palo Alto: Xerox PARC SSL-79-3, July, 1979. 100 pages.

[Christensen 67] Carlos Christensen. "An Example of the Manipulation of Directed Graphs in the AMBIT/G Programming Language," Proceedings of the ACM Symposium on Interactive Systems for Experimental Applied Mathematics, Washington, D.C., August, 1967.

[Christensen 71a] Carlos Christensen. An Introduction to AMBIT/L, a Diagrammatic Language for List Processing. Wakefield, MA: Massachusetts Computer Associates CA-7102-2211, Feb 22, 1971. 65 pages.

[Christensen 71b] Carlos Christensen and Michael Karr. IAM, A System for Interactive Algebraic Manipulation. Wakefield, MA: Massachusetts Computer Associates CA-7103-0311, March 3, 1971. 38 pages.

[Dijkstra 72] Edsger W. Dijkstra. "The Humble Programmer," Communications of the ACM. Vol 15, No. 10, Oct 1972. pp 859-866.

[English 67] W. K. English, D. C. Engelbart, and M. L. Berman. "Display Selection Techniques for Text Manipulation," IEEE Transactions on Human Factors in Electronics. Vol HFE-8, No. 1, March 1967.

[Evans 66] T. G. Evans and D. L. Darley, "On-line Debugging Techniques: A Survey," Proceedings AFIPS Fall Joint Computer Conference. Vol 29, 1966. pp. 37-50.

[Fikes 79] Richard Fikes. Private conversation with the author, Nov. 9, 1979, 1:45 PM.

[Gladwin 69] B. J. Gladwin. "The Utilization of Graphic Display Units as the Main Form of Computer Input," The Computer Journal. Vol 12, No. 2, May 1969. pp. 114-117.

[Gold 69] Michael M. Gold. "Time-Sharing and Batch-Processing: An Experimental Comparison of Their Values in a Problem Solving Situation," Communications of the ACM. Vol 12, No. 5, May 69. pp. 249-259.

[Goldberg 79] A. Goldberg and D. Robson. "A Metaphor for User Interface Design," Proceedings of the 12th Hawaii International Conference on System Sciences 1979. Vol. 1. pp. 148-157.

[Greene 73] R. A. Greene. "Programming Tool For Automated Flow Chart Generation of Assembly Language Programs," IBM Technical Disclosure Bulletin. Vol 15, No. 10, March 1973. pp. 2999-3001.

[Hain, 65] G. Hain and K. Hain. "A general purpose automatic flowcharter," Proc. Fourth Annual Meeting of UAIDE, New York, Oct. 1965. pp. IV-i to IV-12.

[Hanson 78] David R. Hanson. "Event Associations in SNOBOL4 for Program Debugging," Software--Practice and Experience. Vol 8, 1978. pp. 115-129.

[Henderson 69] D. Austin Henderson. A Description and Definition of Simple AMBIT/G--a Graphical Programming Language. Wakefield, MA: Massachusetts Computer Associates CA-6904-2811, April 28, 1969. 32 pages.

[Henderson 79] D. Austin Henderson. Private conversation with the author, Nov. 15, 1979, 2:00 PM.

[Henriksen 77] James O. Henriksen. "An Interactive Debugging Facility for GPSS," IEEE 1977 Winter Simulation Conference. Gaithersburg, MD, Dec 5-7, 1977. pp. 331-8.

[Horning 79] Jim Horning. Private conversation with the author, December 12, 1979, 2:30 PM.

[Hughes 78] T. P. Hughes and D. H. Sawin III. "Breakpoint Design for Debugging Microprocessor Software," Computer Design. Nov 1978. pp. 99-107.

[Ingalls 78] Daniel H. H. Ingalls. "The Smalltalk-76 Programming System Design and Implementation," Fifth Annual ACM Symposium on Principles of Programming Languages. Tucson, Ariz., Jan 23-25, 1978.

[Kahn 72] R. E. Kahn. "Resource-sharing computer communications networks," Proceedings IEEE. Vol. 60, No. 11, November 1972. pp. 1397-1407.

[Kay 69] Alan Curtis Kay. The Reactive Engine. University of Utah Dept of Electrical Engineering and Computer Science PhD Thesis, Aug, 1969. Ann Arbor, Michigan: University Microfilms, Inc., 1972. 327 pages.

[Kay 76] Smalltalk-72 Instruction Manual, Alan Kay and Adele Goldberg, eds. Palo Alto: Xerox PARC SSL-76-6, 1976. 130 pages.

[Kay 77] Alan Kay and Adele Goldberg. "Personal Dynamic Media," IEEE Computer. March 1977. pp. 31-41. An expanded version is available as Palo Alto: Xerox PARC SSL-76-1, March 1976. 74 pages.

[Kazek 78] Chester S. Kazek, Jr. FORTRAN Extended Interactive Debugging Aid. Los Alamos, NM: University of California Los Alamos Scientific Lab LA-7467-MS, Sept, 1978. 11 pages.

[Knuth 63] Donald E. Knuth. "Computer Drawn Flowcharts," Communications of the ACM. Vol 6 No. 9, Sept, 1963. pp. 555-563.

[Knuth 69] Donald E. Knuth. The Art of Computer Programming, Vol 1 Fundamental Algorithms. Reading, Mass: Addison-Wesley Publishing Company, 1969.

[Kotok 61] A. Kotok. DEC Debugging Tape. Cambridge, MA: Massachusetts Institute of Technology Memo MIT-1, December, 1961.

[Laaser 79] Willaim Laaser. Private conversation with the author, Nov. 9, 1979, 2:00 pm.

[Lampson 65] Butler W. Lampson. "Interactive Machine Language Programming," Proc. FJCC, 1965. pp. 473-481.

[Lauesen 75] Soren Lauesen. "A Large Semaphore based operating system," Communications of the ACM. Vol 18, No. 7, July, 1975. pp. 377-389.

[Lauesen 79] Soren Lauesen, "Debugging Techniques," Software--Practice and Experience. Vol 9, 1979. pp. 51-63.

[Lenders 78] P. Lenders and J. Tiberghien. "Debugging Aids for Real Time Microprocessor Systems," Euromicro Journal. Vol 4 No. 4, July 1978. pp. 220-1.

[Leslie 78] John Leslie. "Software Debugging for Beginners," Kilobaud. No. 20, Aug, 1978. pp 40-43.

[Levine 77] Lawrence H. Levine. "Debugging, Planned or Ad Hoc, Which is More Effective?" Journal of Educational Data Processing. Vol 14, No. 4, 1977. pp. 1-9.

[Liskov 77] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. "Abstraction Mechanisms in CLU," Communications of the ACM. Vol 20, No. 8, Aug, 1977. pp. 564-576.

[Loeser 76] R. Loeser and E. M. Gaposchkin. "The second Law of Debugging," Software--Practice and Experience. Vol. 6, 1976. pp. 577-578.

[Lyon 78] Gordon Lyon. Cobol Instrumentation and Debugging: A Case Study. U.S. Dept of Commerce National Bureau of Standards Special Publication 500-26, Jan 1978. 27 pages.

[Metcalfe 76] R. M. Metcalfe and D. R. Boggs. "ETHERNET: Distributed Packet Switching for Local Computer Networks," Communications of the ACM. Vol 19 No. 7, July 1976. pp. 395-404.

[Mitchell 79a] James G. Mitchell. The Design and Construction of Flexible and Efficient Interactive Programming Systems. New York: Garland Publishing, Inc., 1979. 140 pages. Phd Thesis at Carnegie Mellon Univ, 1970.

[Mitchell 79b] James Mitchell, *et. al.* Mesa Language Manual, Version 5.0. Palo Alto: Xerox PARC CSL-79-3, 1979.

[Model 79] Mitchell L. Model. Monitoring System Behavior In a Complex Computational Environment. Palo Alto: Xerox PARC CSL-79-1, January 1979. 179 pages. Also available as Stanford University Computer Science Dept. Report CS-79-701.

[Naur 74] Peter Naur. Concise Survey of Computer Methods. New York: Petrocelli Books, 1974. 307 pages.

[Newman 79] William M. Newman and Robert F. Sproull. Principles of Interactive Computer Graphics, second edition. New York: McGraw-Hill Book Company, 1979. pp. 309-331.

[North 77] Steve North. "A Dynamic Debugging System," Creative Computing. Vol 3, No. 5, Sept-Oct 1977. pp. 26-8.

[Petit 69] Philip Petit. Raid. Stanford Artificial Intelligence Laboratory Operating Note 58, Sept 1969. 24 pages.

[Rovner 69] P. D. Rovner and D. A. Henderson, Jr. "On the Implementation of AMBIT/G: A Graphical Programming Language," Proceedings of the International Joint Conference on Artificial Intelligence. Washington D.C., May 7-9, 1969. pp. 9-19.

[Rovner 79] Paul D. Rovner. Private conversation with the author, Nov 15, 1979, 1:30 PM.

[Sackman 68] H. Sackman, W. J. Erikson and E. E. Grant. "Exploratory Experimental Studies Comparing On-line and off-line Programming Performance," Communications of the ACM. Vol 11, No. 1, Jan 1968. pp 3-11.

[Satterthwaite 75] Edwin H. Satterthwaite, Jr. Source Language Debugging Tools. Stanford University Computer Science Department PhD Thesis Stan-CS-75-494, May 1975. 338 pages.

[Schueler 77] J. Schueler. "Debugging Aids for Assembly Language Programmers," Canadian DataSystems. Vol 9, No. 8, Sept 1977. pp. 37-39.

[Sheppard 79] Sylvia B. Sheppard, Phil Milliman and Bill Curtis. Factors Affecting Programmer Performance in a Debugging Task. Arlington VA: Software management Research Information Systems Programs, General Electric Company TR-79-388100-5, February, 1979. 56 pages.

[Shoch 79] John F. Shoch. "An Overview of the Programming Language Smalltalk-72," ACM Sigplan Notices. Vol 14, No. 9, Sept 1979. pp. 64-73.

[Sutherland 63] Ivan E. Sutherland. Sketchpad: A Man-Machine Graphical Communication System. MIT PhD Thesis. Lexington, MA: Lincoln Labs Technical Report No. 296, Jan 30, 1963, reissued May 19, 1965. 92 pages.

[Sweet 78] Richard Sweet. "Appendix B: Implementation Description," Empirical Estimates of Program Entrophy. Palo Alto: Xerox PARC CSL-78-3, 1978. pp. 85-96.

[Swinehart 74] Daniel Carl Swinehart. Copilot: A Multiple Process Approach to Interactive Programming Systems. Stanford Universtiy Computer Science Department PhD Thesis. SAIL Memo AIM-230 and CSD Report STAN-CS-74-412, July, 1974. 193 pages.

[Teitelman 77] Warren Teitelman. A Display Oriented Programmer's Assistant. Palo Alto: Xerox PARC CSL-77-3, March 8, 1977. 30 pages.

[Teitelman 78] Warren Teitelman. Interlisp Reference Manual. Palo Alto: Xerox PARC, 1978.

[Teitelman 79] Warren Teitelman. Private conversation with the author, Nov. 13, 1979, 12:10 PM.

[Thacker 79] C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs. Alto: A Personal Computer. Palo Alto: Xerox PARC CSL-79-11, August 7, 1979. 50 pages. Paper will also appear in Siewiorek, Bell and Newell, Computer Structures: Readings and Examples, second edition.

[Tratner 79] M. Tratner. "A Fundamental Approach to Debugging," Software--Practice and Experience. Vol 9, 1979. pp. 97-99.

[van Tassel 74] Dennie van Tassel. Program Style, Design, Efficiency, Debugging and Testing. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1974. 256 pages.

[Yarwood 77] Edward Yarwood. Toward Program Illustration. University of Toronto Computer Systems Research Group Technical Report CSRG-84, October 1977 (M. Sc. Thesis).

[Zimmerman 67] Luther L. Zimmerman. "On-line Programming Debugging--A Graphic Approach," Computers and Automation. Vol 16, No. 11, Nov 1967. pp. 30-34.